COMPOSITIONAL REACTIVE SYNTHESIS FOR MULTI-AGENT SYSTEMS

Salar Moarref

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2016

Rajeev Alur, Zisman Family Professor of Computer and Information Science
Ufuk Topcu, Assistant Professor of Aerospace Engineering and Engineering Mechanics
Supervisors of Dissertation

Lyle Ungar, Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee:
Chaired by George Pappas, Professor of Electrical and Systems Engineering
Roderick Bloem, Professor, Graz University, Austria
Vijay Kumar, Professor of Mechanical Engineering and Applied Mechanics
Rahul Mangharam, Associate Professor of Electrical and Systems Engineering

COMPOSITIONAL REACTIVE SYNTHESIS FOR MULTI-AGENT SYSTEMS

To my parents

Acknowledgments

I would like to thank my advisors, Rajeev and Ufuk, for their continual support and guidance throughout my studies, and for providing me with a great deal of freedom for pursuing my ideas and interests, while also kindly and patiently guiding me toward the right direction whenever I needed. This dissertation would not have been possible without their supervision and support.

I thank my dissertation committee members, Roderick Bloem, Vijay Kumar, Rahul Mangharam, and George Pappas, for their comments and feedback that have helped to imrove the quality of this dissertation. I am also grateful to them for being extremely flexible with respect to the scheduling of the dissertation proposal and defense.

I am grateful to my parents for encouraging and supporting my goals and ambitions, and providing me with every oppurtunity to realize my dreams. My stay at Penn was enriched by the company of great collaborators and friends like Ashutosh Trivedi, Zhihao Jiang, Miroslav Pajic, Vojtěch Forejt, Mukund Raghothaman, Abhishek Udupa, Loris D'Antoni, Anduo Wang, Jyotirmoy Deshmukh, Yifei Yuan, Nan Zheng, Kai Hong, Arjun Radhakrishna, Hossein Ahmadzadeh, Shahin Jabbari and Nimit Singhania. I hope that these friendship will continue to grow. Last, but not the least, I would like to express my deepest gratitude to Bi Fei, Mohammad Hassan Lotfi, and Sanaz Ghadiri for always being there for me.

The research described in this dissertation was partially supported by awards NSF Expeditions in Computing CCF 1138996, AFRL FA8650-15-C-2546, ONR N000141310778, ARO W911NF-15-1-0592, NSF 1550212 and DARPA W911NF-16-1-0001.

Abstract

Complex systems often consist of multiple agents (or components) interacting with each other and their environment to perform certain tasks and achieve specified objectives. For example, teams of robots are employed to perform tasks such as monitoring, surveillance, and disaster response in different domains including assembly planning, search and rescue, and object transportation. With growing complexity of systems and guarantees they are required to provide, the need for *automated* and *formal* design approaches that can guarantee safety and correctness of the designed system is becoming more evident. To this end, an ambitious goal in system design and control is to automatically synthesize the system from a high-level specification given in a formal language such as linear temporal logic.

The goal of this dissertation is to investigate and develop the necessary tools and methods for automated synthesis of controllers from high-level specifications for multi-agent systems. We are particularly interested in studying how the existing structure in systems can be exploited to achieve more efficient synthesis algorithms through compositional reasoning. We consider systems where multiple controllable agents react to their environment that can be *dynamically changing* and *potentially adversarial*. The objective of the system is given as a global specification, and the goal is to synthesize controllers for each controlled agent such that the overall system satisfies the specified objective.

We explore three different frameworks for compositional synthesis of controllers for multi-agent systems. In the first framework, we decompose the global specification into local ones, we then refine the local specifications until they become realizable, and we show that under certain conditions, the strategies synthesized for the local specifications guarantee the satisfaction of the global specification. In the second framework, we show how *parametric* and *reactive* controllers can be specified and synthesized, and how they can be automatically composed to enforce a high-level objective. Parameters allow us to take advantage of the symmetry in many synthesis problems, while reactivity of the controllers takes into account that the environment may be dynamic and potentially adversarial. Finally, in the third framework, we focus on a special but practically useful class of multi-agent systems, and show how by taking advantage of the structure in the system and its objective, and through compositional and symbolic synthesis, we can achieve significantly better scalability and can solve problems where the centralized synthesis algorithm is infeasible.

Contents

List of Tables

List of Figures

List of Algorithms

1

Introduction

Complex systems often consist of multiple agents (or components) interacting with each other and their environment to perform certain tasks and achieve specified objectives. For example, teams of robots are employed to perform tasks such as monitoring, surveillance, and disaster response in different domains including assembly planning [HLW00], evacuation [RA10], search and rescue [JWE97], localization [FBKT00], object transportation [RDJ95], and formation control [BA98]. With growing complexity of systems and guarantees they are required to provide, the need for *automated* and *reliable* design and analysis methods and tools is increasing. The necessity becomes more evident considering the safety-critical nature of some of these systems where the consequences of errors can be too catastrophic and even life threatening.

To address these challenges, an emerging trend in systems design and control is to use formal methods, e.g., *model checking*, to ensure the safety and correctness of the designed controllers and guarantee that the system satisfies specified high-level objectives. In model checking, a model of the system is checked exhaustively and automatically for correctness with respect to a given specification. This approach usually involves a design-verify cycle; if verification tool finds a problem, the designer corrects it and runs the verification again. Changing the design to resolve a problem often introduces other problems, causing this cycle to repeat several times until the design is satisfactory.

An alternative and more appealing approach is to automate the design process, i.e., to systematically build the system where the correctness follows from construction. To this end, *reactive synthesis* with the ambitious goal of automatically synthesizing correct-by-construction controllers from high-level specifications, has recently attracted significant attention. Given a specification in a formal language such as linear temporal logic over sets

of input and output signals, the synthesis problem is to find a finite-state reactive system that assigns an output sequence to every possible input sequence such that the resulting computation satisfies the given specification. Intuitively, the input signals are those that are uncontrollable, i.e., system has no control over their values, contrary to output signals whose values are decided by the system. The synthesis problem can be viewed as a game between two players, the system and its environment. The goal of synthesis is to construct a finite-state system that satisfies the specification regardless of how its environment behaves.

Unfortunately, high complexity of synthesis procedures has restricted the application to relatively small-sized problems. The pioneering work by Pnueli et. al [PR89] showed that reactive synthesis from linear temporal logic specifications is intractable. This high computational burden has prohibited the practitioners from utilizing automated synthesis algorithms. Nevertheless, recent advances in this growing research area have enabled automatic synthesis of interesting real-world systems [BJP+12], indicating the potential of the synthesis algorithms for solving realistic problems. The key insight is to consider more restricted yet practically useful subclasses of the general problem, and in this dissertation we take a step toward this direction.

The goal of this dissertation is to investigate and develop the necessary tools and methods for automated synthesis of controllers from high-level specifications for multi-agent systems. We are particularly interested in studying how the existing structure in such problems can be exploited to achieve more efficient synthesis algorithms, e.g., through compositional synthesis techniques. We are interested in systems where a set of controlled agents react to their environment that includes other uncontrolled, *dynamic* and *potentially adversarial* agents. The objective of the system is given as a global specification, and the goal is to synthesize controllers for each controlled agent such that the overall system satisfies the specified objectives.

To this end, we explore different frameworks for compositional synthesis of controllers for multi-agent systems. The general problem is, given a multi-agent system consisting of controllable (cooperative) and uncontrollable (adversarial) agents, and a global objective specified in temporal logic, how we can synthesize controllers for each controllable agent such that the resulting system satisfies the given objective. The overall theme of the solution approaches is to take advantage of the existing structure in multi-agent systems in order to decompose the synthesis problem into smaller and more manageable subproblems, solving

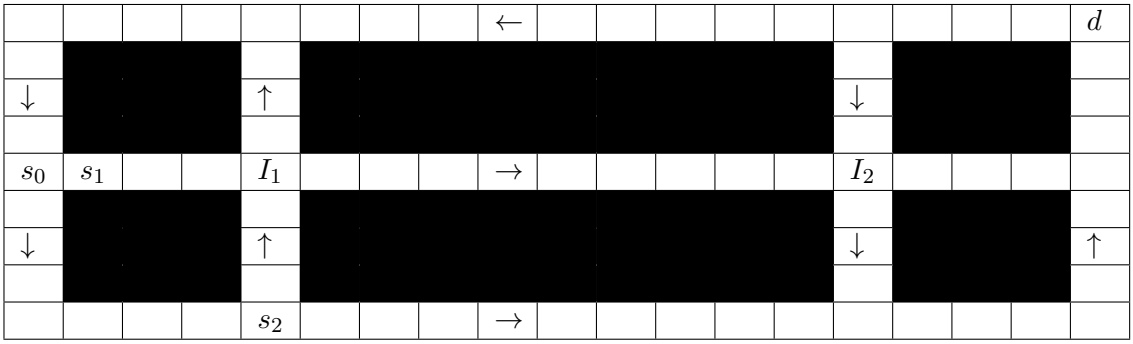


Figure 1.1: One-way streets connected by intersections.

the subproblems, and merging the results to obtain a solution to the main problem. Next we present a brief summary of each proposed framework within this dissertation. A more thorough exposition will be provided in the subsequent chapters.

1.1 Pattern-Based Assume-Guarantee Synthesis

Compositional synthesis techniques can potentially address the scalability problem by solving the synthesis problem for smaller components and merging the results such that the composition satisfies the specification. The challenge is then to find proper decompositions and assumptions-guarantees such that each component is realizable, its expectations of its environment can be discharged on the environment and other components, and circular reasoning is avoided, so that the local controllers can be implemented simultaneously and their composition satisfies the original specification.

In *pattern-based assume-guarantee synthesis* framework, we consider a system with two controllable agents reacting to their dynamically changing and adversarial environment. We decompose the global specification into two local specifications, one for each agent. We refine the local specifications by automatic synthesis of assumptions and guarantees through analysis of strategies and counter-strategies obtained for the agents' local specifications. We show how behaviors of the environment and the system can be inferred from counter-strategies and strategies, respectively, as formulas in special forms called *patterns*. Local specifications are refined until both become realizable, and under certain conditions, the strategies synthesized for the local specifications guarantee the satisfaction of the global specification. Intuitively, additional assumptions and guarantees synthesized during the refinement process are "contracts" between the agents that allow each of them to compute a

strategy for its local specification while ensuring the satisfaction of the global specification for the system.

Example 1.1. *Consider the network of one-way roads divided into grids as shown in Figure 1.1. Assume there are two autonomous vehicles V_1 and V_2 that starting from locations s_0 and s_2, respectively, must reach and cross the intersection I_1 without collision with each other. For simplicity, assume each vehicle can either stop and stay at the same grid, or move one step forward in the road's specified direction. We can write these requirements as a temporal logic specification. In the first framework, we assume that the global specification is decomposed into local specifications, one for each controlled agent. A possible decomposition is as follows. The local specification for vehicle V_1 requires it to eventually reach and cross the intersection. That is, V_1 has no knowledge of V_2. The local specification for V_2 requires it to reach and cross the intersection, and also to avoid collision with V_1. The synthesized controllers for these local specifications will also guarantee satisfaction of the global specification.*

A controller for V_1 that can satisfy its local specification can be automatically synthesized. For example, V_1 can keep moving forward until it reaches and then crosses the intersection. However, the local specification for the vehicle V_2 is unrealizable*, i.e., there is no controller that can satisfy it. From the perspective of V_2, V_1 can stop at the intersection forever as it is "allowed" in V_2's local specification. In the first framework, such counterexamples are analyzed and a set of additional assumptions and guarantees are automatically synthesized. For example, an assumption that V_1 always eventually leaves the intersection can be synthesized and added to V_2's local specification. It is also checked that V_1 can indeed guarantee this assumption. This way, the first framework iteratively discovers and refines the local specifications until all of them become realizable.*

1.2 Compositional Synthesis with Parametric Reactive Controllers

In practice, complex systems are often not constructed from scratch but from a set of existing building blocks. For example in robot motion planning, a robot usually has a number of predefined motion primitives that can be selected and composed to enforce a high-level objective. Intuitively, a compositional approach that solves smaller and more manageable

subproblems, and hierarchically composes the solutions to implement more complicated behaviors seems to be a more plausible way to synthesize complex systems.

We propose a compositional and hierarchical framework for synthesis from a library of *parametric* and *reactive* controllers. Parameters allow us to take advantage of the symmetry in many synthesis problems, e.g., in motion planning for autonomous robots and vehicles. Reactivity of the controllers takes into account that the environment may be dynamic and potentially adversarial. We show how these controllers can be synthesized from parametric objectives specified by the user to form a library of parametric and reactive controllers. We then give a synthesis algorithm that selects and instantiates controllers from the library in order to satisfy a given safety and reachability objective. To show the potential of our framework, we implement and apply the methods to an autonomous vehicle case study, where a controller is synthesized from a library of parametric and reactive controllers to safely navigate a controlled vehicle to its destination while avoiding collision with other uncontrolled vehicles.

Example 1.2. *Consider the network of one-way roads shown in Figure 1.1. Assume there is a controlled vehicle V_1 that must safely navigate from its initial location s_0 to its destination d. Safe navigation means that V_1 must obey the traffic rules (e.g., move in the specific direction of the road) and avoid collision with static obstacles and other uncontrolled vehicles.*

In the second framework, we can take advantage of the symmetry in the problem to synthesize parametric and reactive controllers. Assume x and y are two variables indicating the location of V_1. Let a and b be two parameters. The user can specify a controller C_1 that starting from the parametric state $(x, y) = (a, b)$, eventually advances the vehicle three steps toward east, i.e., eventually $(x, y) = (a + 3, b)$, while avoiding collision with other dynamic and uncontrolled vehicle. Similarly, a parametric and reactive controller C_2 can be synthesized that advances the vehicle two steps toward north while avoiding collision with other vehicles. Synthesized controllers are the building blocks for the compositional algorithm proposed in the second framework. The composer automatically selects and instantiates the controllers from a given library to enforce the high-level objective. For example, to navigate V_1 from its initial location to its destination, the composer can consecutively instantiate and apply C_1 to safely navigate V_1 to the right-most column, and then consecutively apply C_2 to move V_1 toward north until it finally reaches its destination.

1.3 Compositional Synthesis for Decoupled Multi-Agent Systems

In assume-guarantee synthesis framework described above, systems with multiple components can be treated in a *decentralized* manner by considering one component as a part of the environment of another component. However, in this synthesis approach it is difficult to capture and model the need for joint decision-making and cooperative objectives. To address this difficulty, we propose a compositional framework for a special class of multi-agent systems (inspired by decentralized control and swarm robotics literature) based on automatic decomposition of objectives and compositional reactive synthesis using maximally permissive strategies [FJR11]. In this approach, we assume that the objective of the system is given in a conjunctive form. We make an observation that in many cases, each conjunct of the global objective only refers to a small subset of agents in the system. We take advantage of this structure to decompose the synthesis problem: for each conjunct of the global objective, we only consider the agents that are involved, and compute the maximally permissive strategies for those agents with respect to the considered conjunct. We then intersect the strategies to remove potential conflicts between them, and project back the constraints to subproblems. The subproblems are solved again with updated constraints, and this process is repeated until the strategies reach a fixed point. With this approach we manage to solve synthesis problems for systems with multiple agents and objectives such as collision avoidance, formation control and reachability, and for grid-worlds of sizes that are much larger than the cases considered in similar works in the related literature. We show that the compositional algorithm outperforms the centralized synthesis approach, both from time and memory perspective, and can solve problems where the centralized algorithm is infeasible.

Example 1.3. *Consider the network of one-way roads shown in Figure 1.1. Assume there are two controlled autonomous vehicles V_1 and V_2 initially at positions s_0 and s_1. Suppose the objective of the system is for V_1 and V_2 to infinitely visit locations d and s_0 while obeying traffic rules, avoiding collision with each other, with static obstacles and also with other uncontrolled vehicles. Furthermore, assume V_1 and V_2 are required to stay close to each other, e.g., they must not be more than two steps away from each other. The latter requirement needs cooperation and joint decision-making between V_1 and V_2. In the third framework,*

we show how such requirements can be specified and propose a compositional and symbolic algorithm for synthesizing controllers for controlled agents.

1.4 Contributions

We now provide a short summary of the contributions made by this dissertation[1]:

- We propose algorithms for automatic refinement of temporal logic specifications by synthesizing additional environment assumptions or system guarantees. The suggested refinements can be validated by the user to ensure compatibility with her design intent.
- We propose three different approaches that can be used to refine the specifications of the components in the context of compositional synthesis. Intuitively, these approaches differ in how much information about one component is shared with the other one. We show that providing more knowledge of one component's behavior for the other component can make it significantly easier to refine the local specifications, with the expense of increasing the coupling between the components. We illustrate and compare the methods with examples and a case study.
- We give a symbolic algorithm for synthesizing a control strategy that reactively chooses and instantiates controllers from a given library of controllers to enforce a high-level safety and reachability objective for the system. We show how a designer can simply specify parametric controllers and then a controller and its interface along with acceptable parameter values are synthesized automatically. We implement our algorithms symbolically using binary decision diagrams (BDDs) and apply them to an autonomous vehicle case study to show the potential of our approach.
- We propose a framework for modular specification and compositional controller synthesis for multi-agent systems with *imperfect* controlled agents, i.e., agents that can only partially observe the state of the system. We give a compositional algorithm that automatically decomposes the synthesis problem using the structure in the system and compositionally synthesizes controllers for the agents. We implement the methods symbolically using BDDs and apply them to a robot motion planning case study. We report on our experimental results and show that the compositional algorithm can significantly outperform the centralized approach.

[1] All the results appearing in this dissertation are published in [AMT13, AMT15, AMT16, AMT].

The organization of the subsequent chapters is as follows. Chapter 2 introduces some notation, background and definitions that are used in the rest of the dissertation. Proposed frameworks are described in chapters 3, 4 and 5. These chapters are written in a way that can be read independently from each other. Chapter 6 provides an overview of the related work in the area and discusses how the proposed methods described in this manuscript differs from earlier work. Finally, Chapter 7 concludes this dissertation by summarizing its contributions and highlighting some of the potential future directions.

2

Preliminaries

In this chapter we introduce some notation, background and definitions that are used in the rest of the dissertation. Let $\mathbb{Z}$ be the set of integers. For $a, b \in \mathbb{Z}$, let $[a..b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$.

Linear temporal logic (LTL). We use LTL to specify system objectives. LTL is a formal specification language with two types of operators: logical connectives (e.g., $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), and $\rightarrow$ (implication)) and temporal operators (e.g., and $\Box$ (always), $\bigcirc$ (next), $\mathcal{U}$ (until), $\Diamond$ (eventually)). Let $\mathcal{V}$ be a finite set of Boolean variables (atomic propositions). The formulas of LTL are defined over a set of atomic propositions $\mathcal{V}$. The syntax is given by the grammar:

$$\Phi := v \mid \Phi \vee \Phi \mid \neg\Phi \mid \bigcirc \Phi \mid \Phi \,\mathcal{U}\, \Phi \quad \text{for } v \in \mathcal{V}$$

We define $\texttt{true} = v \vee \neg v$, $\texttt{false} = v \wedge \neg v$, $\Diamond\Phi = \texttt{true}\,\mathcal{U}\,\Phi$, and $\Box\Phi = \neg\Diamond\neg\Phi$. A formula with no temporal operator is a Boolean formula or a *predicate*. Given a predicate ϕ over variables $\mathcal{V}$, we say $s \in 2^{\mathcal{V}}$ satisfies ϕ, denoted by $s \models \phi$, if the formula obtained from ϕ by replacing all variables in s by `true` and all other variables by `false` is valid. Formally, we define $s \models \phi$ inductively as

- for $v \in \mathcal{V}$, $s \models v$ if and only if $v \in s$,
- $s \not\models \phi$ if and only if $s \not\models \phi$, and
- $s \models \phi \vee \psi$ if and only if $s \models \phi$ or $s \models \psi$.

We call the set of all possible assignments to variables $\mathcal{V}$ *states* and denote them by $\Sigma_{\mathcal{V}}$, i.e., $\Sigma_{\mathcal{V}} = 2^{\mathcal{V}}$. An LTL formula over variables $\mathcal{V}$ is interpreted over infinite words $w \in (\Sigma_{\mathcal{V}})^{\omega}$. The language of an LTL formula Φ, denoted by $\mathcal{L}(\Phi)$, is the set of infinite words that satisfy

Φ, i.e., $\mathcal{L}(\Phi) = \{w \in (\Sigma_\mathcal{V})^\omega \mid w \models \Phi\}$, where the satisfaction relation $w = \sigma_0\sigma_1\sigma_2\cdots \models \Phi$ is inductively defined as follows:

1. $w \models v$ if $v \in \sigma_0$,
2. $w \models \Phi_1 \vee \Phi_2$ if $w \models \Phi_1$ or $w \models \Phi_2$,
3. $w \models \neg\Phi$ if $w \not\models \Phi$,
4. $w \models \bigcirc\Phi$ if $\sigma_1\sigma_2\cdots \models \Phi$, and
5. $w \models \Phi_1 \mathcal{U} \Phi_2$ if there is $n \geq 0$ such that $\sigma_n\sigma_{n+1}\cdots \models \Phi_2$ and for all $0 \leq i < n, \sigma_i\sigma_{i+1}\cdots \models \Phi_1$.

Example 2.1. *Let $\mathcal{V} = \{r, c, g, v\}$ be a set of Boolean variables. Here r, c, g and v stand for* request*,* clear*,* grant *and* valid *signals, respectively. Consider the following LTL formulas: $\Phi_1 = \Box(r \rightarrow \bigcirc\Diamond g)$, $\Phi_2 = \Box((c \vee g) \rightarrow \bigcirc\neg g)$, $\Phi_3 = \Box(c \rightarrow \neg v)$ and $\Phi_4 = \Box\Diamond(g \wedge v)$. Intuitively, Φ_1 requires that every request must be granted eventually starting from the next step by setting signal g to high. Φ_2 says that if clear or grant signal is high, then grant must be low at the next step. Φ_3 says if clear is high, then the valid signal must be low. Finally, Φ_4 says that system must issue a valid grant infinitely often.*

We often use predicates over $\mathcal{V} \cup \mathcal{V}'$ where $\mathcal{V}'$ is the set of primed versions of the variables in $\mathcal{V}$, i.e., $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. Given a subset of variables $\mathcal{X} \subseteq \mathcal{V}$ and a state $s \in \Sigma_\mathcal{V}$, we denote by $s_{|\mathcal{X}}$ the projection of s to $\mathcal{X}$, i.e., $s_{|\mathcal{X}} = \{x \in \mathcal{X} \mid x \in s\}$. For a predicate ϕ over variables $\mathcal{V}$, we let $[\![\phi]\!]$ be the set of valuations over $\mathcal{V}$ that make ϕ true, that is, $[\![\phi]\!] = \{s \in \Sigma_\mathcal{V} \mid s \models \phi\}$. For a set $\mathcal{Z} \subseteq \mathcal{V}$, let $Same(\mathcal{Z}, \mathcal{Z}')$ be a predicate specifying that the value of the variables in $\mathcal{Z}$ stay unchanged during a transition. Ordered binary decision diagrams (OBDDs) can be used for obtaining concise representations of sets and relations over finite domains [CGP99]. If R is an n-ary relation over $\{0, 1\}$, then R can be represented by the BDD for its *characteristic function*:

$$f_R(x_1, \cdots, x_n) = 1 \text{ if and only if } R(x_1, \cdots, x_n) = 1.$$

With a little bit abuse of notation and when it is clear from the context, we treat sets and functions as their corresponding predicates.

Generalized Reactivity (1) (GR(1)). Let $\mathcal{V}$ be a set of Boolean variables partitioned into input $\mathcal{I}$ and output $\mathcal{O}$ variables. GR(1) specifications are of the form $\Phi = \Phi_e \rightarrow \Phi_s$, where Φ_α for $\alpha \in \{e, s\}$ can be written as a conjunction of the following parts:

1	2
3	4

Figure 2.1: A small grid-world

- ϕ_i^α: A predicate over $\mathcal{I}$ if $\alpha = e$ and over $\mathcal{I} \cup \mathcal{O}$ otherwise, characterizing the initial state.
- Φ_g^α: A formula of the form $\bigwedge_i \square\lozenge B_i$ characterizing fairness/liveness, where each B_i is a predicate over $\mathcal{I} \cup \mathcal{O}$.
- Φ_t^α: An LTL formula of the form $\bigwedge_i \square\psi_i$ characterizing safety and transition relations, where ψ_i is a predicate over expressions v and $\bigcirc v'$ where $v \in \mathcal{I} \cup \mathcal{O}$ and, $v' \in \mathcal{I}$ if $\alpha = e$ and $v' \in \mathcal{I} \cup \mathcal{O}$ if $\alpha = s$.

Observe that GR(1) is a fragment of LTL. Intuitively, Φ_e indicates the assumptions on the environment and Φ_s characterizes the requirements of the system. Any correct implementation that satisfies the specification guarantees to satisfy Φ_s, provided that the environment satisfies Φ_e.

Example 2.2. *Consider the* 2×2 *grid-world shown in Figure 2.1. Assume there are two robots* R_1 *and* R_2 *initially at locations* 1 *and* 4*, respectively. We use variables* $X_1, X_2 \in [1..4]$ *to encode the location of each robot at any time-step. At each time-step, each robot can move from its current location to one of its neighboring cells by moving up, down, left or right. Assume* R_1 *is controlled while* R_2 *is not, i.e.,* X_1 *is the output variable and* X_2 *is the input variable. The goal of* R_1 *is to always eventually visit the cell* 3*, and it must also avoid being at the same cell as* R_1 *at any time-step. We assume that* R_2 *infinitely often visits the cell* 3*.*

We can formally specify above requirements in GR(1) as follows. The predicates $\phi_i^e := X_2 = 4$ *and* $\phi_i^s := X_1 = 1$ *specify the initial locations of* R_2 *and* R_1*, respectively. The formula*

$$\Phi_t^i = \square(X_i = 1 \rightarrow \bigcirc(X_i = 2 \vee X_i = 3)) \wedge \square(X_i = 2 \rightarrow \bigcirc(X_i = 1 \vee X_i = 4)) \wedge$$
$$\square(X_i = 3 \rightarrow \bigcirc(X_i = 1 \vee X_i = 4)) \wedge \square(X_i = 4 \rightarrow \bigcirc(X_i = 2 \vee X_i = 3))$$

for $i \in \{1, 2\}$ *characterizes the transitions of robot* R_i*. The formula* $\Phi_s^1 = \square(X_1 \neq X_2)$ *indicates that* R_1 *must not be at the same cell with* R_2*. The formula* $\Phi_g^1 = \square\lozenge(X_1 = 3)$ *characterizes the goal of* R_1*. Similarly, the formula* $\Phi_g^2 = \square\lozenge(X_2 = 3)$ *characterizes the liveness assumption about* R_2*. Finally, the GR(1) specification* $\Phi = \Phi_e \rightarrow \Phi_s$ *encodes the*

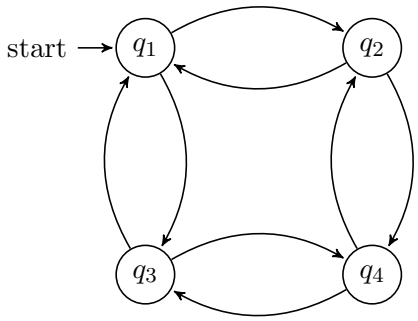


Figure 2.2: An LTS $\mathcal{T}$

system requirements where $\Phi_e = \phi_i^2 \wedge \Phi_t^2 \wedge \Phi_g^2$ *encodes assumptions on the environment (robot* R_1*,) and* $\Phi_s = \phi_i^1 \wedge \Phi_t^1 \wedge \Phi_s^1 \wedge \Phi_g^1$ *characterizes system guarantees.*

Labeled Transition System (LTS). An LTS is a tuple $\mathcal{T} = \langle Q, Q_0, \delta, \mathcal{L} \rangle$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta \subseteq Q \times Q$ is a transition relation, and $\mathcal{L} : Q \to \phi$ is a labeling function that maps each state to a predicate ϕ over variables $\mathcal{V}$. Without loss of generality, we assume that every state of an LTS has an outgoing edge, i.e., for all $q \in Q$ there exists $q' \in Q$ such that $(q, q') \in \delta$. A *run* of an LTS is an infinite sequence of states $\sigma = q_0 q_1 q_2 ...$ where $q_0 \in Q_0$ and for any $i \geq 0$, $q_i \in Q$ and $(q_i, q_{i+1}) \in \delta$. The language of an LTS $\mathcal{T}$ is defined as the set $\mathcal{L}(\mathcal{T}) = \{w \in Q^\omega \mid w \text{ is a run of } \mathcal{T}\}$, i.e., the set of infinite words generated by the runs of $\mathcal{T}$. We often consider an LTS as a directed graph with a natural bijection between the states and transitions of the LTS and vertices and edges of the graph, respectively. Formally for an LTS $\mathcal{T} = \langle Q, Q_0, \delta, \mathcal{L} \rangle$, we define the graph $\mathcal{G}_\mathcal{T} = \langle V, E \rangle$ where each $v_i \in V$ corresponds to a unique state $q_i \in Q$, and $(v_i, v_j) \in E$ if and only if $(q_i, q_j) \in \delta$.

Example 2.3. *Consider the setting introduced in Example 2.2. We can model the transitions of robot* R_1 *with an LTS* $\mathcal{T} = \langle Q, Q_0, \delta, \mathcal{L} \rangle$ *where* $Q = \{q_1, q_2, q_3, q_4\}$*,* $Q_0 = \{q_1\}$*,* $\delta = \{(q_1, q_2), (q_1, q_3), (q_2, q_1), (q_2, q_4), (q_3, q_1), (q_3, q_4), (q_4, q_2), (q_4, q_3)\}$*, and the labeling function* $\mathcal{L}$ *is defined as* $\mathcal{L}(q_i) = (X_1 = i)$ *for* $i \in [1..4]$*, i.e., each state of* $\mathcal{T}$ *corresponds to a possible location for* R_1 *in the grid-world. Figure 2.2 shows the graphical representation of the LTS* $\mathcal{T}$*.*

Moore (Mealy) Transducer. A *Moore transducer* is a tuple $M = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \gamma)$, where S is a set of states, $s_0 \in S$ is an initial state, $\Sigma_\mathcal{I} = 2^\mathcal{I}$ is the input alphabet, $\Sigma_\mathcal{O} = 2^\mathcal{O}$ is the output alphabet, $\delta : S \times \Sigma_\mathcal{I} \to S$ is a transition function and $\gamma : S \to \Sigma_\mathcal{O}$ is a state

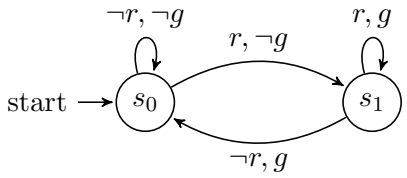


Figure 2.3: A Mealy transducer

output function. A *Mealy* transducer is similar, except that the state output function is $\gamma : S \times \Sigma_{\mathcal{I}} \to \Sigma_{\mathcal{O}}$. For an infinite word $w \in (\Sigma_{\mathcal{I}})^{\omega}$, a run of M is an infinite sequence $\sigma \in S^{\omega}$ such that $\sigma_0 = s_0$ and for all $i \geq 0$, $\sigma_{i+1} = \delta(\sigma_i, w_i)$. The run σ on input word w produces an infinite word $M(w) \in (\Sigma_{\mathcal{V}})^{\omega}$ such that $M(w)_i = \gamma(\sigma_i) \cup w_i$ for all $i \geq 0$. The language of M is the set $\mathcal{L}(M) = \{M(w) \mid w \in \mathcal{I}^{\omega}\}$ of infinite words generated by runs of M.

Example 2.4. *Let r and g be two Boolean variables representing* request *and* grant *signals, respectively. Figure 2.3 shows a Mealy transducer $M = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \gamma)$ where $S = \{s_0, s_1\}$, $\mathcal{I} = \{r\}$, $\mathcal{O} = \{g\}$, the transition function δ is defined as $\delta(s_0, \{r\}) = s_1$, $\delta(s_0, \{\}) = s_0$, $\delta(s_1, \{r\}) = s_1$, and $\delta(s_1, \{\}) = s_0$, and the state output function γ is defined as $\gamma(s_0, \{r\}) = \{\}$, $\gamma(s_0, \{\}) = \{\}$, $\gamma(s_1, \{r\}) = \{g\}$, and $\gamma(s_1, \{\}) = \{g\}$. Intuitively, everytime the request signal is high, the mealy transducer M produces a grant at the next step.*

LTL Realizability and Synthesis. An LTL formula Φ is *satisfiable* if there exists an infinite word $w \in (\Sigma_{\mathcal{V}})^{\omega}$ such that $w \models \Phi$. A Moore (Mealy) transducer M satisfies an LTL formula Φ, written as $M \models \phi$, if and only if $\mathcal{L}(M) \subseteq \mathcal{L}(\phi)$. An LTL formula Φ is *Moore (Mealy) realizable* if there exists a Moore (Mealy, respectively) transducer M such that $M \models \Phi$. The *realizability problem* asks whether there exists such a transducer for a given Φ. Given an LTL formula Φ over variables $\mathcal{V}$ and a partitioning of $\mathcal{V}$ into $\mathcal{I}$ and $\mathcal{O}$, the *synthesis problem* is to find a Mealy transducer M with input alphabet $\Sigma_{\mathcal{I}} = 2^{\mathcal{I}}$ and output alphabet $\Sigma_{\mathcal{O}} = 2^{\mathcal{O}}$ that satisfies Φ. A *counter-strategy* for the synthesis problem is a strategy for the environment that can falsify the specification, no matter how the system plays. Formally, a counter-strategy can be represented by a Moore transducer $M_c = (S^c, s_0^c, \mathcal{I}^c, \mathcal{O}^c, \delta^c, \gamma^c)$ that satisfies $\neg\Phi$, where $\mathcal{I}^c = \mathcal{O}$ and $\mathcal{O}^c = \mathcal{I}$ are the input and output variables for M_c which are generated by the system and the environment, respectively.

Game structures. A game structure $\mathcal{G}$ of imperfect information is a tuple $\mathcal{G} = (\mathcal{V}, \Lambda, \tau, \mathcal{OBS}, \gamma)$ where $\mathcal{V}$ is a finite set of variables, Λ is a finite set of actions, τ is a predicate over $\mathcal{V} \cup \Lambda \cup \mathcal{V}'$ defining $\mathcal{G}$'s transition relation, $\mathcal{OBS}$ is a finite set of observable

variables, and $\gamma : \Sigma_{\mathcal{OBS}} \to 2^{\Sigma_{\mathcal{V}}} \setminus \emptyset$ maps each observation to its corresponding set of states. We assume that the set $\{\gamma(o) \mid o \in \Sigma_{\mathcal{OBS}}\}$ partitions the state space $\Sigma_{\mathcal{V}}$[2]. A game structure $\mathcal{G}$ is called *perfect information* if $\mathcal{OBS} = \mathcal{V}$ and $\gamma(s) = \{s\}$ for all $s \in \Sigma_{\mathcal{V}}$. We omit $(\mathcal{OBS}, \gamma)$ in the description of games of perfect information.

Within the scope of this dissertation, we consider two-player turn-based game structures where player-1 and player-2 take turn playing. Let $t \in \mathcal{V}$ be a special variable with domain $\{1, 2\}$ determining which player's turn it is during the game. Without loss of generality, we assume that player-1 always starts the game unless specified otherwise. For $i = 1, 2$, let $\Sigma^i_{\mathcal{V}} = \{s \in \Sigma_{\mathcal{V}} \mid s_{|t} = i\}$ denote player-i's states in the game structure. At any state $s \in \Sigma^i_{\mathcal{V}}$, player-$i$ chooses an action $\ell \in \Lambda$ such that there exists a successor state $s' \in \Sigma_{\mathcal{V}'}$ where $(s, \ell, s') \models \tau$. Intuitively, at a player-i state, she chooses an available action according to the transition relation τ and the next state of the system is chosen from the possible successor states. For every state $s \in \Sigma_{\mathcal{V}}$, we define $\Gamma(s) = \{\ell \in \Lambda \mid \exists s' \in \Sigma_{\mathcal{V}'}.\ (s, \ell, s') \models \tau\}$ to be the set of available actions at that state. A *run* in $\mathcal{G}$ from an initial state $s_{init} \in \Sigma_{\mathcal{V}}$ is a sequence of states $\pi = s_0 s_1 s_2 \cdots$ such that $s_0 = s_{init}$ and, for all $i > 0$, there is an action $\ell_i \in \Lambda$ with $(s_{i-1}, \ell_i, s'_i) \models \tau$, where s'_i is obtained by replacing the variables in s_i by their primed copies. A run π is maximal if either it is infinite or it is finite and ends in a state $s \in \Sigma_{\mathcal{V}}$ where $\Gamma(s) = \emptyset$. The *observation sequence* of π is the unique sequence $Obs(\pi) = o_0 o_1 o_2 \cdots$ such that for all $i \geq 0$, we have $s_i \in \gamma(o_i)$. For $\ell \in \Lambda$ and $X \subseteq \Sigma_{\mathcal{V}}$, let $Post^{\mathcal{G}}_{\ell}(X) = \{r \in \Sigma_{\mathcal{V}} \mid \exists s \in X \ :\ (s, \ell, r') \models \tau\}$. *Composition* of two game structures $\mathcal{G}_1 = (\mathcal{V}^1, \Lambda^1, \tau^1), \mathcal{G}_2 = (\mathcal{V}^2, \Lambda^2, \tau^2)$ of perfect information, denoted by $\mathcal{G}^{\otimes} = \mathcal{G}_1 \otimes \mathcal{G}_2$, is a game structure $\mathcal{G}^{\otimes} = (\mathcal{V}^{\otimes}, \Lambda^{\otimes}, \tau^{\otimes})$ of perfect information where $\mathcal{V}^{\otimes} = \mathcal{V}^1 \cup \mathcal{V}^2$, $\Lambda^{\otimes} = \Lambda^1 \cup \Lambda^2$, and $\tau^{\otimes} = \tau^1 \wedge \tau^2$.

Example 2.5. *Let $t \in \{1, 2\}$ and $x \in [0..4]$ be two variables. Figure 2.4 shows an explicit representation of a two-player turn-based game structure $\mathcal{G}$ of perfect information where player-1 (player-2) states are represented with ovals (boxes, repsectively). The game structure $\mathcal{G}$ is defined over variables $\mathcal{V} = \{t, x\}$ and actions $\Lambda = \{inc, dec\}$. Intuitively, at any player-i state for $i \in \{1, 2\}$, she chooses an available action to increment or decrement the value of x. Note that at player-2 states with $x \in [0..2]$ if she chooses the action inc, the value of x can be incremented non-deterministically by one or two. Also note that inc action is*

[2]This assumption can be weakened to a covering of the state space where observations can overlap [CDHR06, DWDR06].

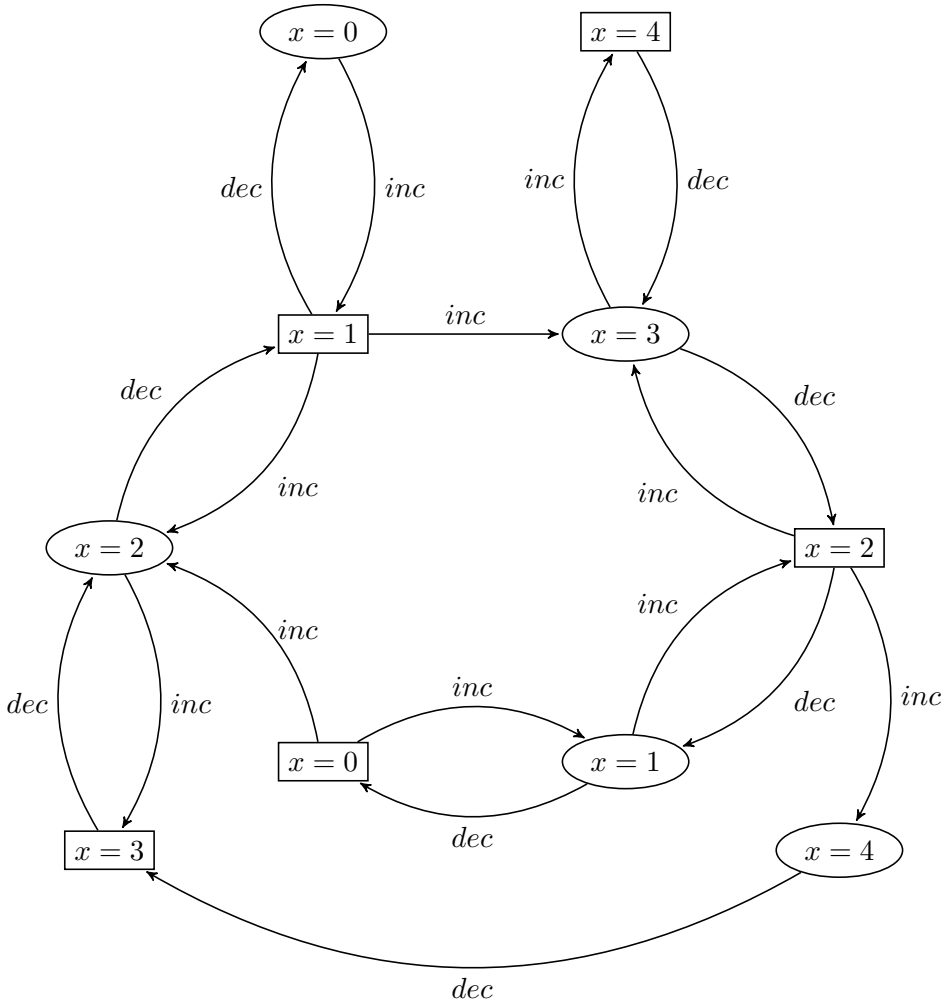


Figure 2.4: A game structure $\mathcal{G}$ defined over a variable $x \in [0..4]$.

not available at player-2 states with $x \in [3..4]$. *The transition relation* τ *of* $\mathcal{G}$ *is defined symbolically as* $\tau = \tau_e \wedge \tau_s$ *where*

$$\tau_e := t = 1 \wedge t' = 2 \wedge \bigwedge_{i=1}^{4} (x = i \wedge dec \wedge x' = i - 1) \wedge \bigwedge_{i=0}^{3} (x = i \wedge inc \wedge x' = i + 1), \text{ and}$$

$$\tau_s := t = 2 \wedge t' = 1 \wedge \bigwedge_{i=1}^{4} (x = i \wedge dec \wedge x' = i - 1) \wedge \bigwedge_{i=0}^{2} (x = i \wedge inc \wedge (x' = i + 1 \vee x' = i + 2)).$$

Intuitively, τ_e (τ_s) defines player-1 (player-2, respectively) transitions.

Strategies. A *strategy* $\mathtt{S}$ in $\mathcal{G}$ for player-i, $i \in \{1,2\}$, is a function $\mathtt{S} : (\Sigma_{\mathcal{V}})^*.\Sigma^i_{\mathcal{V}} \to \Lambda$. A strategy $\mathtt{S}$ in $\mathcal{G}$ for player-2 is *observation-based* if for all prefixes $\rho_1, \rho_2 \in (\Sigma_{\mathcal{V}})^*.\Sigma^2_{\mathcal{V}}$, if $Obs(\rho_1) = Obs(\rho_2)$, then $\mathtt{S}(\rho_1) = \mathtt{S}(\rho_2)$. We are interested in the existence of observation-based strategies for player-2. Given two strategies $\mathtt{S}_1$ and $\mathtt{S}_2$ for player-1 and player-2, respectively, the *possible outcomes* $\Omega_{\mathtt{S}_1,\mathtt{S}_2}(s)$ from a state $s \in \Sigma_{\mathcal{V}}$ are runs: a run $s_0 s_1 s_2 \cdots$ belongs to $\Omega_{\mathtt{S}_1,\mathtt{S}_2}(s)$ if and only if $s_0 = s$ and for all $j \geq 0$ either s_j has no successor, or $s_j \in \Sigma^i_{\mathcal{V}}$ and $(s_j, \mathtt{S}_i(s_0 \cdots s_j), s'_{j+1}) \models \tau$.

Strategies may need memory to remember the history of a game. Let M be a finite set called *memory*. A finite-memory strategy $\mathtt{S} = (m_0, f_M, f_\Lambda)$ for player-i is defined as an initial memory $m_0 \in M$ along with a pair of functions: a memory-update function $f_M : M \times \Sigma_{\mathcal{V}} \to M$, which given the current state of the game and the memory, updates the memory, and a next-action function $f_\Lambda : M \times \Sigma^i_{\mathcal{V}} \to \Lambda$, which given the current player-i state and the memory, suggests the next action for the player. A strategy $\mathtt{S}$ is memory-less (a.k.a. positional) if the memory M is a singleton, i.e., $|M| = 1$. A memory-less strategy is independent of the history of the game and only depends on the current state. Thus, a memory-less strategy for player-i can be represented as a function $\mathtt{S} : \Sigma^i_{\mathcal{V}} \to \Lambda$.

Winning condition. A game $(\mathcal{G}, \phi_{init}, \Phi)$ consists of a game structure $\mathcal{G}$, a predicate ϕ_{init} specifying a set of initial states, and an LTL objective Φ for player-2. A run $\pi = s_0 s_1 \cdots$ is winning for player-2 if it is infinite and $\pi \in \mathcal{L}(\Phi)$. Let Π be the set of runs that are winning for player-2. A strategy $\mathtt{S}_2$ is winning for player-2 if for all strategies $\mathtt{S}_1$ of player-1 and all initial states $s_{init} \models \phi_{init}$, we have $\Omega_{\mathtt{S}_1,\mathtt{S}_2}(s_{init}) \subseteq \Pi$, that is, all possible outcomes are winning for player-2. It is well known that for ω-regular objectives, the games are determined, i.e., either player-2 has a winning strategy or player-1 has a spoiling strategy [GH82]. We say the game $(\mathcal{G}, \phi_{init}, \Phi)$ is realizable if and only if the system has a winning strategy in the game $(\mathcal{G}, \phi_{init}, \Phi)$.

Knowledge game structure. For a game structure $\mathcal{G} = (\mathcal{V}, \Lambda, \tau, \mathcal{OBS}, \gamma)$ of imperfect information, a game structure $\mathcal{G}^K$ of perfect information can be obtained using a subset construction procedure such that for any objective Φ, there exists a deterministic observation-based strategy for player-2 in $\mathcal{G}$ with respect to Φ if and only if there exists a deterministic winning strategy for player-2 in $\mathcal{G}^K$ for Φ [Rei84, CDHR06]. Formally, we define the *knowledge-based subset construction* of $\mathcal{G}$ as the game structure $\mathcal{G}^K = (\mathcal{V}^K, \Lambda, \tau^K)$ of perfect

information where $\mathcal{V}^K = 2^{\mathcal{V}} \setminus \{\emptyset\}$ and $(s_1, \ell, s_2) \in \tau^K$ iff there exists $obs \in \mathcal{OBS}$ such that $s_2 = Post^{\mathcal{G}}_{\ell}(s_1) \cap \gamma(obs)$ and $s_2 \neq \emptyset$. Intuitively, each state in $\mathcal{G}^K$ is a set of states of $\mathcal{G}$ that represents player-2's knowledge about the possible states in which the game can be after a sequence of observations. In the worst case, the size of $\mathcal{G}^K$ is exponentially larger than the size of $\mathcal{G}$. We refer to $\mathcal{G}^K$ as the *knowledge* game structure corresponding to $\mathcal{G}$. In the rest of this chapter, we only consider game structures of perfect information.

Solving games. Let $\mathcal{G} = (\mathcal{V}, \Lambda, \tau)$ be a game structure of perfect information. Let $\varphi = \exists\Lambda\ \exists\mathcal{V}'.\tau$ be a predicate specifying the set of states in $\mathcal{G}$ with at least one outgoing transition, i.e., the set of states $s \in \Sigma_{\mathcal{V}}$ for which there exists an action $\ell \in \Lambda$ and next state $s' \in \Sigma_{\mathcal{V}'}$ such that $(s, \ell, s') \models \tau$. The set $\mathcal{D} = \Sigma_{\mathcal{V}} \setminus [\![\varphi]\!]$ of dead-end states, i.e., states with no outgoing transition, can be computed symbolically as $\varphi_{\mathcal{D}} = \neg\varphi$. That is, $\mathcal{D} = [\![\varphi_{\mathcal{D}}]\!]$. Note that any dead-end state is losing for player-2 by definition.

The operator $Epre_{\Lambda} : 2^{\Sigma_{\mathcal{V}}} \to 2^{\Sigma_{\mathcal{V}}}$ maps a set $X \subseteq \Sigma_{\mathcal{V}}$ of states to the states for which there exists an action $\ell \in \Lambda$ such that all ℓ-successors belong to the set X, and is formally defined as follows:

$$\begin{aligned} Epre_{\Lambda}(X) &= \{v \in \Sigma_{\mathcal{V}} \mid (\exists \ell \in \Lambda \forall w \in \Sigma_{\mathcal{V}}.(v, \ell, w') \models \tau \rightarrow w \in X) \wedge v \models \varphi\} \\ &= (\exists\Lambda\forall\mathcal{V}'.(\tau \rightarrow X')) \wedge \varphi. \end{aligned}$$

Note that the set of dead-end states are excluded from $Epre_{\Lambda}(X)$ by conjoining the pre-image states with the predicate φ.

The operator $Apre_{\Lambda} : 2^{\Sigma_{\mathcal{V}}} \to 2^{\Sigma_{\mathcal{V}}}$ maps a set $X \subseteq \Sigma_{\mathcal{V}}$ of states to the states for which for all actions $\ell \in \Lambda$ all ℓ-successors belong to the set X, and is formally defined as

$$\begin{aligned} Apre_{\Lambda}(X) &= \{v \in \Sigma_{\mathcal{V}} \mid (\forall \ell \in \Lambda \forall w \in \Sigma_{\mathcal{V}}.(v, \ell, w') \models \tau \rightarrow w \in X) \wedge v \models \varphi\} \\ &= (\forall\Lambda\forall\mathcal{V}'.(\tau \rightarrow X')) \wedge \varphi. \end{aligned}$$

Symbolic algorithms for solving the realizability and synthesis problems are based on the *controllable predecessor* operator [MPS95]. The (player-2) controllable predecessor operator $CPre : 2^{\Sigma_{\mathcal{V}}} \to 2^{\Sigma_{\mathcal{V}}}$ maps a set $X \subseteq \Sigma_{\mathcal{V}}$ of states to the states from which player-2 can force the game into X in one step. Player-2 can force the game into X from a state $s \in \Sigma^1_{\mathcal{V}}$ iff for all available moves ℓ, all ℓ-successors of s are in X, and she can force the game into X from a state $s \in \Sigma^2_{\mathcal{V}}$ iff there is *some* available action ℓ such that all ℓ-successors of v are in X.

Formally,

$$CPre(X) = (t = 2 \wedge Epre_{\Lambda}(X)) \vee (t = 1 \wedge Apre_{\Lambda}(X)).$$

The set of states from which player-2 can avoid a set $[\![\Phi_{err}]\!] \subseteq \Sigma_{\mathcal{V}}$ of states is the greatest fixed point $\nu X.[\![\neg\Phi_{err}]\!] \cap CPre(X)$ (safety objective,) and the set of states from which player-2 can reach a set $[\![\Phi_{reach}]\!] \subseteq \Sigma_{\mathcal{V}}$ of states is the least fixed point $\mu X.[\![\Phi_{reach}]\!] \cup CPre(X)$ (reachability objective). Roughly speaking, the fixed point algorithm that computes the set $\mathcal{W}$ of winning states over the game structure $\mathcal{G}$ and with respect to a safety or reachability objective, iteratively computes sets of states $\mathcal{W}_i$ for $i \geq 0$ until it reaches the fixed point $\mathcal{W}_i = \mathcal{W}_{i+1} = \mathcal{W}$.

Safety Games. In Chapter 5, we use the bounded synthesis approach [SF07a, FJR11] to solve the synthesis problems from LTL specifications. In [FJR11], it is shown how LTL formulas can be reduced to *safety games*. Formally, a safety game is a game $(\mathcal{G}, \phi_{init}, \Phi)$ with a special safety objective $\Phi = \Box(\texttt{true})$. That is, any infinite run in the game structure $\mathcal{G}$ starting from any initial state $s \models \phi_{init}$ is winning for player-2. We drop Φ from description of safety games as it is implicitly defined. Intuitively, in a safety game, the goal of player-2 is to avoid the dead-end states, i.e., states that there is no available action. We refer the readers to [FJR11, Ehl12] for details of reducing LTL formulas to safety games and solving them. To solve a game $(\mathcal{G}, \phi_{init}, \Phi)$ using bounded synthesis approach, we first obtain the game structure $\mathcal{G}^{\Phi}$ corresponding to Φ using the methods proposed in [FJR11], and then solve the safety game $(\mathcal{G} \otimes \mathcal{G}^{\Phi}, \phi_{init})$ to determine the winner of the game and compute a winning strategy for player-2, if one exists.

Maximally permissive strategies. Safety games are *memory-less determined*, i.e., player-2 wins the game if and only if there exists a strategy $\mathtt{S} : \Sigma^2_{\mathcal{V}} \to \Lambda$. Intuitively, a memory-less strategy only depends on the current state and is independent from the history of the game. Let $(\mathcal{G}, \phi_{init})$ be a safety game, where $\mathcal{G} = (\mathcal{V}, \Lambda, \tau)$ is a game structure of perfect information. Assume $W \subseteq \Sigma_{\mathcal{V}}$ be the set of winning states for player-2, i.e., from any state $s \in W$ there exists a strategy $\mathtt{S}_2$ such that for any strategy $\mathtt{S}_1$ chosen by player-1, all possible outcomes $\pi \in \Omega_{\mathtt{S}_1,\mathtt{S}_2}(s)$ are winning. The *maximally permissive strategy* $\mathcal{S} : \Sigma^2_{\mathcal{V}} \to 2^{\Lambda}$ for player-2 is defined as follows: for all $s \in \Sigma^2_{\mathcal{V}}$, $\mathcal{S}(s) = \{\ell \in \Lambda \mid \forall r \in \Sigma_{\mathcal{V}'}.\ (s, \ell, r) \models \tau_s \to r \in W\}$, i.e., the set of actions ℓ where all ℓ-successors belong to the set of winning states. It is well known that $\mathcal{S}$ subsumes all winning strategies of player-2 in the safety game $(\mathcal{G}, \Phi_{init})$. Composition of two maximally permissive strategies $\mathcal{S}_1, \mathcal{S}_2 : \Sigma^2_{\mathcal{V}} \to 2^{\Lambda}$, denoted by $\mathcal{S} = \mathcal{S}_1 \otimes \mathcal{S}_2$, is defined

as $\mathcal{S}(s) = \mathcal{S}_1(s) \cap \mathcal{S}_2(s)$ for any $s \in \Sigma_{\mathcal{V}}$, i.e., the set of allowed actions by $\mathcal{S}$ at any state $s \in \Sigma_{\mathcal{V}}$ is the intersection of the allowed actions by $\mathcal{S}_1$ and $\mathcal{S}_2$. The restriction of the game structure $\mathcal{G}$ with respect to its maximally permissive strategy $\mathcal{S}$ is the game structure $\mathcal{G}[\mathcal{S}] = (\mathcal{V}, \Lambda, \tau \wedge \phi_{\mathcal{S}})$ where $\phi_{\mathcal{S}}$ is the predicate encoding $\mathcal{S}$, i.e., for all $(s, \ell) \in \Sigma_{\mathcal{V}}^2 \times \Lambda$, $(s, \ell) \models \phi_{\mathcal{S}}$ if and only if $\ell \in \mathcal{S}(s)$. Intuitively, $\mathcal{G}[\mathcal{S}]$ is the same as $\mathcal{G}$ but player-2's actions are restricted according to $\mathcal{S}$.

3

Pattern-Based Assume-Guarantee Synthesis

The reactive synthesis problem is known to be intractable for general LTL specifications [Ros92]. However, there are fragments of LTL, such as Generalized Reactivity(1), for which the realizability and synthesis problems can be solved in polynomial time in the size of the state space [BJP+12]. Yet scalability is a big challenge as increasing the number of formulas in a specification may cause an exponential blowup in the size of its state space [BJP+12]. Compositional synthesis techniques are used to address this issue by solving the synthesis problem for smaller components and merging the results such that the composition satisfies the specification. The challenge is then to find proper decompositions and interface specifications such that each component is realizable, its expectations of its environment can be discharged on the environment and other components, and circular reasoning is avoided, so that the local controllers can be implemented simultaneously and their composition satisfies the original specification [OTM11].

In this chapter, we study the problem of synthesizing interface specifications between components in the context of compositional reactive synthesis. To this end, we consider a problem in which the system consists of two components C_1 and C_2 and that a global specification is given which is realizable and decomposed into two local specifications, corresponding to C_1 and C_2, respectively. We consider a special case in which there is a serial interconnection between the components [OTM11], i.e., roughly speaking, the dependency between components' output variables is acyclic and only the output variables of C_2 depend on the output variables of C_1. We are interested in computing refinements such that the refined local specifications are both realizable and when implemented, the resulting system satisfies the global specification.

Our solution is based on automated refinement of assumptions and guarantees expressed

in LTL. The core of the method is the synthesis of a set of LTL formulas of special form, called patterns, which hold over all runs of an abstraction of the strategy or counter-strategy computed for the specification. If the local specification for a component C_2 is unrealizable, we refine its environment assumptions, while ensuring that the other component C_1 can indeed guarantee those assumptions. To this end, it is sometimes necessary to refine C_1's specification by adding guarantees to it. We propose three different approaches that can be used to synthesize the interface specifications of the components in the context of compositional synthesis. Intuitively, these approaches differ in how much information about one component is shared with the other one. We show that providing more knowledge of one component's behavior for the other component can make it significantly easier to refine the interface specifications, with the expense of increasing the coupling between the components. We illustrate and compare the methods with examples and a case study.

3.1 Overview and Problem Statement

Assume a global LTL specification is given that is realizable. Furthermore, assume the system consists of a set of components, and that a decomposition of the global specification into a set of local ones is given, where each local specification corresponds to a system component. The decomposition may result in components whose local specifications are unrealizable, e.g., due to the lack of adequate assumptions on their environment. The general question is how to refine the local specifications such that the refined specifications are all realizable, and when implemented together, the resulting system satisfies the global specification.

We consider a special case of this problem. We assume the system consists of two components C_1 and C_2, where there is a serial interconnection between the components [OTM11]. Intuitively, it means that the dependency between the output variables of the components is acyclic, as shown in Figure 3.1. This assumption enables us to define a total order over the components and avoid circular reasoning. Let $\mathcal{I}$ be the set of input variables controlled by the environment and $\mathcal{O}$ be the set of output variables controlled by the system, partitioned into $\mathcal{O}_1$ and $\mathcal{O}_2$, the set of output variables controlled by C_1 and C_2, respectively. For a specification $\Phi = \Phi_e \rightarrow \Phi_s$, we define an *assumption refinement* $\Psi_e = \bigwedge_i \Psi_{e_i}$ as a conjunction of a set of environment assumptions such that $(\Phi_e \wedge \Psi_e) \rightarrow \Phi_s$ is realizable. Similarly, $\Psi_s = \bigwedge_i \Psi_{s_i}$ is a *guarantee refinement* if $\Phi_e \rightarrow (\Phi_s \wedge \Psi_s)$ is realizable.

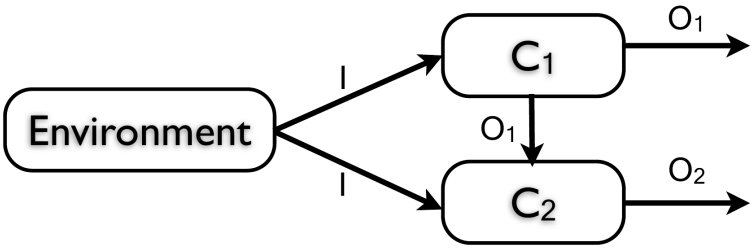


Figure 3.1: Serial interconnection.

An assumption refinement Ψ_e is consistent with Φ if $\Phi_e \wedge \Psi_e$ is satisfiable. Note that an inconsistent refinement $\Phi_e \wedge \Psi_e = \texttt{false}$ leads to an specification which is trivially realizable, but neither interesting nor useful.

We now formally define the problem that is considered in this chapter.

Problem Statement 3.1. *Consider a realizable global specification $\Phi = \Phi_e \rightarrow \Phi_s$. Assume Φ is decomposed into two local specifications $\Phi_1 = \Phi_{e_1} \rightarrow \Phi_{s_1}$ and $\Phi_2 = \Phi_{e_2} \rightarrow \Phi_{s_2}$ such that $\Phi_e \rightarrow (\Phi_{e_1} \wedge \Phi_{e_2})$ and $(\Phi_{s_1} \wedge \Phi_{s_2}) \rightarrow \Phi_s$. We assume Φ_e, Φ_s, Φ_{e_1}, Φ_{s_1}, Φ_{e_2}, and Φ_{s_2} are LTL formulas which only contain variables from the sets $\mathcal{I}$, $\mathcal{I} \cup \mathcal{O}$, $\mathcal{I}$, $\mathcal{I} \cup \mathcal{O}_1$, $\mathcal{I} \cup \mathcal{O}_1$, and $\mathcal{I} \cup \mathcal{O}$, respectively. We would like to find refinements Ψ and Ψ' such that the refined specifications $\Phi_1^{ref} = \Phi_{e_1} \rightarrow (\Phi_{s_1} \wedge \Psi')$ and $\Phi_2^{ref} = (\Phi_{e_2} \wedge \Psi) \rightarrow \Phi_{s_2}$ are both realizable, and $\Psi' \rightarrow \Psi$.*

If refinements Ψ and Ψ' exist, then the resulting system from implementing the refined specifications Φ_1^{ref} and Φ_2^{ref} satisfies the global specification Φ [OTM11]. We use this fact to establish the correctness of the decomposition and refinements in our proposed approaches. As Φ is realizable, and C_1 is independent from C_2, it follows that Φ_1 (in case it is not realizable) can be made realizable by adding assumptions on its environment. Especially, providing all the environment assumptions of the global specification for C_1 is enough to make its specification realizable. However, this might not be the case for Φ_2. In the rest of this chapter, we assume that Φ_1 is realizable, while Φ_2 is not. We investigate how the strategy and counter-strategy computed for C_1 and C_2, respectively, can be used to find suitable refinements for the interface specifications.

Our solution is based on an automated refinement of assumptions and guarantees expressed in LTL. We refine an unrealizable specification by adding assumptions on its environment. The refinement is synthesized step by step guided by counter-strategies. When the specification is unrealizable, a counter-strategy is computed and a set of formulas of

Algorithm 3.1: FindGuarantees

Input: $\Phi = \Phi_e \rightarrow \Phi_s$: a realizable specification, U: subset of variables
Output: $\mathcal{P}$: A set of formulas Ψ such that $\Phi_e \rightarrow (\Phi_s \wedge \Psi)$ is realizable

```
1 M_s = ComputeStrategy(Φ);
2 P := Infer-GR(1)-Formulas(M_s, U);
3 P' := Infer-Complement-GR(1)-Formulas(M_s, U);
4 foreach Ψ ∈ P' do
5     if (Φ_e → (Φ_s ∧ ¬Ψ)) is realizable then
6         P = P ∪ ¬Ψ ;
7 return P ;
```

the forms $\Diamond\Box\psi$, $\Diamond\psi$, and $\Diamond(\psi \wedge \bigcirc\psi')$, which hold over *all* runs of the counter-strategy, is inferred. Intuitively, these formulas describe potentially "bad" behaviors of the environment that may cause unrealizability. Their complements (which are in forms allowed by GR(1) syntax) form the set of *candidate* assumptions, and adding any of them as an assumption to the specification prevents the environment player (player-1) from behaving according to the counter-strategy (without violating its assumptions). We say the counter-strategy is ruled out from the environment's possible behaviors. Counter-strategy-guided refinement algorithm (explained in detail in Section 3.3) iteratively chooses and adds a candidate assumption to the specification, and the process is repeated until the specification becomes realizable, or the search cannot find a refinement within the specified search depth. The user is asked to specify a subset of variables to be used in synthesizing candidate assumptions. This subset may reflect the designer's intuition on the source of unrealizability, and help to narrow down the search for finding a proper refinement.

A similar idea can be used to refine the guarantees of a specification. When the specification is realizable, a winning strategy can be computed for the system. The winning strategy might not be unique, that is, there may be several strategies that can satisfy the same specification. The computed strategy for a realizable specification restricts the possible runs of the system to the ones which satisfy the given specification. As it is deterministic, it might also put different restrictions on the system. These restrictions define the differences between two winning strategies that satisfy the same specification.

We can use patterns to infer the behaviors of the strategies as LTL formulas. The inferred formulas can be used in two ways. One is to get an insight into the possible behaviors and additional guarantees that a given strategy provide. They can also be used to restrict the

1	2	3	4
5	6	7	8

Figure 3.2: Room in Example 3.1

system by adding guarantees, similar to restricting the environment by adding assumptions. Restricting the system can be used to compute a different winning strategy which satisfies the original specification, and also provides additional guarantees.

Formulas of the form $\square\lozenge\psi$, $\square\psi$, and $\square(\psi \rightarrow \bigcirc\psi')$ can be used to infer *implicit* guarantees provided by the given strategy, i.e., they can be added to the original specification as guarantees, and the same strategy satisfies the new specification as well as the original one. These formulas can be seen as additional guarantees a component can provide in the context of compositional synthesis. Formulas of the form $\lozenge\square\psi$, $\lozenge\psi$, and $\lozenge(\psi \wedge \bigcirc\psi')$ can be used to restrict the system by adding the complement of them to the specifications as guarantees. As a result, the current strategy is ruled out from system's possible strategies and therefore, the new specification, if still realizable, will have a different strategy which satisfies the original specification, and also provides additional guarantees. Algorithm 3.1 shows how a set of additional guarantees $\mathcal{P}$ are computed for the specification Φ and subset of variables U. For the computed strategy $\mathcal{M}_s$, the procedure **Infer-GR(1)-Formulas** synthesizes formulas of the forms $\square\lozenge\psi, \square\psi$, and $\square(\psi \rightarrow \bigcirc\psi')$ which hold over all runs of the strategy. Similarly, the procedure **Infer-Complement-GR(1)-Formulas** synthesizes formulas of the form $\lozenge\square\psi, \lozenge\psi$, and $\lozenge(\psi \wedge \bigcirc\psi')$. These procedures are explained in Section 3.2. In what follows, we will use *grid-world* examples commonly used in robot motion planning case studies to illustrate the concepts and techniques [LaV06].

Example 3.1. *Assume there are two robots, R_1 and R_2, in a room divided into eight cells as shown in Figure 3.2. Both robots must infinitely often visit the goal cell 4. Besides, they cannot be in the same cell simultaneously (no collision). Finally, at any time-step, each robot can either stay put or move to one of its neighbor cells. In the sequel, assume i ranges over $\{1, 2\}$. We denote the location of robot R_i with Loc_{R_i}, and cells by their numbers. Initially $Loc_{R_1} = 1$ and $Loc_{R_2} = 8$.*

The global specification is realizable. Note that in this example, all the variables are controlled and there is no external environment. Assume that the specification is decomposed into Φ_1 and Φ_2, where $\Phi_i = \Phi_{e_i} \rightarrow \Phi_{s_i}$ is the local specification for R_i. Assume $\Phi_{e_1} =$ `true`,

i.e., no assumption on the environment of R_1, and Φ_{s_1} only includes the initial location of R_1, its transition rules, and its goal to infinitely often visit cell 4. Φ_{s_2} includes the initial location of R_2, its transition rules, its objective to infinitely often visit cell 4, while avoiding collision with R_1. Here R_1 serves as the environment for R_2 which can play adversarially. Assume Φ_{e_2} only includes the initial location of R_1.

***Inferring formulas**: Φ_1 is realizable. A winning strategy $\mathcal{M}_{S_1}$ for R_1 is to move to cell 2 from the initial location, then to cell 3, and then to move back and forth between cells 4 and 3 forever. The following are examples of formulas inferred from this strategy:*

- *eventually always: $\Diamond\Box(Loc_{R_1} \in \{3,4\})$,*
- *eventually: $\Diamond(Loc_{R_1} = 3)$, $\Diamond(Loc_{R_1} = 4)$,*
- *eventually next: $\Diamond(Loc_{R_1} = 3 \wedge \bigcirc Loc_{R_1} = 4)$, $\Diamond(Loc_{R_1} = 4 \wedge \bigcirc Loc_{R_1} = 3)$,*
- *always eventually: $\Box\Diamond(Loc_{R_1} = 3)$, $\Box\Diamond(Loc_{R_1} = 4)$,*
- *always: $\Box(Loc_{R_1} \in \{1,2,3,4\})$, and*
- *always next: $\Box(Loc_{R_1} = 2 \rightarrow \bigcirc Loc_{R_1} = 3)$, $\Box(Loc_{R_1} = 3 \rightarrow \bigcirc Loc_{R_1} = 4)$.*

***Refining assumptions**: Note that Φ_2 includes no assumption on R_1 other than its initial location. Specifically, Φ_2 does not restrict the way R_1 can move. That is, from the perspective of R_2, R_1 can go to any cell at any time-step. The specification Φ_2 is unrealizable. A counter-strategy for R_1 is to move from cell 1 to the goal cell 4, and stay there forever, preventing R_2 from fulfilling its requirements. Using counter-strategy-guided refinement for refining the assumptions on the environment, we find the refinements $\Psi_1 = \Box\Diamond(Loc_{R_1} \neq 4)$, $\Psi_2 = \Box(Loc_{R_1} \neq 4)$, and $\Psi_3 = \Box(Loc_{R_1} = 4 \rightarrow \bigcirc Loc_{R_1} \neq 4)$. Ψ_1 states that R_1 should infinitely often move out of the goal location. Ψ_2 says that R_1 must never enter the goal location. Ψ_3 says that if R_1 is at the goal location, it must move out of it at the next time-step. Intuitively, these refinements suggest that R_1 is not present at cell 4 at some point during the execution. Adding any of these formulas to the assumptions of Φ_2 makes it realizable. The designer can validate and choose the appropriate refinement.*

***Refining guarantees**: Formula $\Psi_4 = \Diamond\Box(Loc_{R_1} \in \{3,4\})$ is satisfied by $\mathcal{M}_{S_1}$, meaning that R_1 eventually reaches and stays at the cells 3 and 4 forever. An example of a guarantee refinement is to add the guarantee $\neg\Psi_4 = \Box\Diamond(Loc_{R_1} \notin \{3,4\})$ to Φ_1, meaning that the robot R_1 should infinitely often move out of cells 3 and 4. A winning strategy for the new specification is to move back and forth in the first row between initial and goal cells. That is, R_1 has the infinite run $(1,2,3,4,3,2)^\omega$.*

We use these techniques to refine the interface specifications. We propose three different approaches for finding suitable refinements, based on how much information about the strategy of the realizable component is allowed to be shared with the unrealizable component. The first approach has no knowledge of the strategy chosen by C_1, and tries to find a refinement by analyzing counter-strategies. The second approach iteratively extracts some information from the strategies computed for Φ_1, and uses this information to refine the specifications. The third approach encodes the strategy as a conjunction of LTL formulas, and provides it as a set of assumptions for C_2, allowing it to have a full knowledge of the strategy. These approaches are explained in detail in Section 3.4.

Compositional Refinement: Assume $\mathcal{M}_{S_1}$ is the computed strategy for R_1. The first approach, computes a refinement for the unrealizable specification, then checks if the other component can guarantee it. For example, Ψ_3 is a candidate refinement for Φ_2, i.e., by assuming that R_1 infinitely often leaves the goal cell, there exists a strategy for R_2 to satisfy its objective. Now we need to ensure that this assumption is indeed guaranteed by R_1 in order to ensure that the global specification is satisfied. The strategy $\mathcal{M}_S$ can provide such a guarantee. However, there exists other winning strategies that can satisfy ϕ_1 which do not guarantee Ψ_3. For example, if R_1 reaches the goal location and stays there forever, Φ_1 is still satisfied. However, this strategy does not satisfy Ψ_3 anymore. To ensure that the specification for R_1 satisfies Ψ_3 no matter how the strategy is computed, Φ_1 should be refined. Φ_1 can be refined by Ψ_3 added to its guarantees. The strategy $\mathcal{M}_{S_1}$ still satisfies the new specification, and refined specifications are both realizable. Thus, the first approach returns Ψ_3 as a possible refinement.

Using the second approach, formula $\Psi_5 = \square\lozenge(Loc_{R_1} = 3)$ is inferred from $\mathcal{M}_{S_1}$. Refining both specifications with Ψ_5 leads to two realizable specifications, hence Ψ_4 is returned as a refinement. The third approach encodes $\mathcal{M}_{S_1}$ as conjunction of transition formulas $\Psi_6 = \bigwedge_{i=1}^{3} \square(Loc_{R_1} = i \rightarrow \bigcirc Loc_{R_1} = i + 1) \wedge \square(Loc_{R_1} = 4 \rightarrow \bigcirc Loc_{R_1} = 3)$. Refining assumptions of Φ_2 with Ψ_6 makes it realizable.

3.2 Inferring Behaviors as LTL Formulas

In this section we show how certain types of LTL formulas that hold over all runs of a counter-strategy or strategy can be synthesized. The user chooses the subset U of variables to be used in synthesizing the formulas of each kind. These formulas are obtained in the

following manner. First an LTS $\mathcal{T}$ is obtained from the given Moore (Mealy) transducer $\mathcal{M}$ which represents the counter-strategy (strategy, respectively). Next, using the set U, an abstraction $\mathcal{T}^a$ of $\mathcal{T}$ is constructed which is also an LTS. A set of patterns which hold over all runs of $\mathcal{T}^a$ is then synthesized. The instantiations of these patterns form the set of formulas which hold over all runs of the input transducer. Next we explain these steps in more detail.

3.2.1 Constructing the Abstract LTS

We now provide the details of obtaining an abstract LTS from a given Moore (Mealy) transducer $\mathcal{M} = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \gamma)$. Without loss of generality we assume that all incoming transitions of a state $s_i \in S$ of a Moore transducer have the same truth valuation over input and output propositions. That is, if $s_i, s_j, s_k \in S$ are states of $\mathcal{M}$, and $\gamma(s_i) = out$, $\delta(s_i, in) = s_k$, $\gamma(s_j) = out'$, and $\delta(s_j, in') = s_k$ for $in, in' \in \Sigma_{\mathcal{I}}$ and $out, out' \in \Sigma_{\mathcal{O}}$, then we have $in = in'$ and $out = out'$. We make a similar assumption about Mealy transducers. This way each state of the $\mathcal{M}$ would have a unique label, and we define a labeling function $\Gamma_{\mathcal{M}} : S \to \Sigma_{\mathcal{V}}$ which maps each state $s \in S$ to a set of propositions that hold in s.

Given the Moore (Mealy) transducer $\mathcal{M}$ of the counter-strategy (strategy, respectively), first an LTS $\mathcal{T} = (Q, \{q_0\}, \delta_{\mathcal{T}}, \mathcal{L})$ is obtained which keeps the structure of $\mathcal{M}$ while removing its input and output details. Formally, for each $s_i \in S$, there is a unique $q_i \in Q$, and $q_0 \in Q$ is the state corresponding to $s_0 \in S$. There exists a transition between $q_i, q_j \in Q$, i.e., $(q_i, q_j) \in \delta_{\mathcal{T}}$ if and only if there exists some input $\sigma \in \Sigma_{\mathcal{I}}$ such that $\delta(s_i, \sigma) = s_j$. We let $\mathcal{L}(q_i) = \Gamma_{\mathcal{M}}(s_i)$ for each state $q_i \in Q$. That is, the label $\mathcal{L}(q_i)$ is consistent with the truth values of the input and output variables in the state $s_i \in S$.

Using the subset $U \subseteq \mathcal{I} \cup \mathcal{O}$ of variables chosen by the user, an abstraction $\mathcal{T}^a = (Q^a, Q_0^a, \delta_{\mathcal{T}^a}, \mathcal{L}^a)$ of $\mathcal{T}$ is computed based on the state labels $\mathcal{L}$. There is a surjective function $F : Q \to Q^a$ which maps each state of $\mathcal{T}$ to a unique state of $\mathcal{T}^a$. Intuitively, the abstraction $\mathcal{T}^a$ has a unique state for each maximal subset of states of $\mathcal{T}$ which have the same projected labels, and if there is a transition between two states of $\mathcal{T}$, there will be a transition between their mapped states in $\mathcal{T}^a$. Formally, there exists a unique state $q^u \in Q^a$ for any $u \in \Sigma_U$ corresponding to the maximal set $Q_{q^u} = \{q \in Q \mid \mathcal{L}(q)_{|U} = u\}$, i.e., the maximal subset of states of $\mathcal{T}$ that have the same projected label. For all $q^u \in Q^a$, we define $F^{-1}(q^u) = Q_{q^u}$, and $\mathcal{L}(q^u) = u$. The initial state q_0^a of $\mathcal{T}^a$ is defined as $Q_0^a = \{F(q_0) = q_0^a\}$. Finally, there is

a transition between $q_i^a, q_j^a \in Q^a$, i.e., $\delta^a(q_i^a) = q_j^a$ if and only if there exist $q_i, q_j \in Q$ such that $F(q_i) = q_i^a$, $F(q_j) = q_j^a$, and $\delta(q_i) = q_j$. The following theorem states that $\mathcal{T}^a$ simulates $\mathcal{T}$. Therefore, any LTL formula Φ that is satisfied by $\mathcal{T}^a$ is also satisfied by $\mathcal{T}$.

Theorem 3.1. *For any two states $p \in Q^a$ and $q \in Q$, if $F(q) = p$ then* (1) $\mathcal{L}(p) = \mathcal{L}(q)_{|U}$, (2) *for any $q' \in Q$ where $q' \in \delta_{\mathcal{T}}(q)$, there is a state $p' \in Q^a$ such that $p' \in \delta_{\mathcal{T}^a}(p)$ and $F(q') = p'$.*

Proof. (1) easily follows from the construction. We prove that the mapping function F defines a simulation relation between states of $\mathcal{T}$ and $\mathcal{T}^a$, that is, $\mathcal{T} \preceq_F \mathcal{T}^a$. Assume there are states $p \in Q$ and $q \in Q^a$ such that $F(p) = q$. Assume there is a state $p' \in Q$ such that $\delta(p) = p'$, that is, there is a transition from p to p'. By definition of $\mathcal{T}^a$, there should be a state $q' \in Q^a$, such that $F(p') = q'$ and $\delta^a(q) = q'$. □

3.2.2 Synthesizing Patterns

Next we discuss how patterns of certain types can be synthesized from a given LTS $\mathcal{T}$. A pattern $\psi_{\mathcal{P}}$ is an LTL formula which is satisfied over all runs of $\mathcal{T}$, i.e., $\mathcal{T} \models \psi_{\mathcal{P}}$. We are interested in patterns of the forms $\Diamond\Box\psi_{\mathcal{P}}$, $\Diamond\psi_{\mathcal{P}}$, $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$, $\Box\Diamond\psi_{\mathcal{P}}$, $\Box\psi_{\mathcal{P}}$, and $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$, where $\psi_{\mathcal{P}}$ and $\psi'_{\mathcal{P}}$ are propositional formulas expressed as a disjunction of subset of states of $\mathcal{T}$. Patterns are synthesized using graph search algorithms that search for special *configurations*. For an LTS $\mathcal{T} = (Q, Q_0, \delta, \mathcal{L})$, a configuration $C \subseteq Q$ is a subset of states of $\mathcal{T}$. A configuration C is a $\bowtie$-configuration where $\bowtie \in \{\Box, \Box\Diamond, \Diamond, \Diamond\Box\}$ if $\mathcal{T} \models \bowtie \bigvee_{q \in C} q$. For example, C is an $\Box\Diamond$-*configuration* if any run of $\mathcal{T}$ always eventually visits a state from C. A $\bowtie$-configuration C is minimal, if there is no configuration $C' \subset C$ which is an $\bowtie$-configuration, i.e., removing any state from C leads to a configuration which is not a $\bowtie$-configuration anymore. Minimal $\bowtie$-configurations are interesting since they lead to the *strongest* patterns of $\bowtie$-form. Formally, given two non-equivalent predicates ϕ_1 and ϕ_2, we say ϕ_1 is *stronger* than ϕ_2 if $\phi_1 \rightarrow \phi_2$ holds. We will come back to this point in Section 3.3.

Synthesizing $\Box\Diamond\psi_{\mathcal{P}}$ Patterns

To compute all minimal always eventually patterns one can enumerate the configurations, each time pick a configuration and remove its corresponding states from the input LTS, and check whether there exists a cycle in the remaining transition system. However, the problem

of finding all minimal always eventually patterns is NP-hard as stated in the following theorem.

Theorem 3.2. *Computing all minimal $\square\Diamond$-configurations is NP-hard.*

Proof. We give a reduction from the hitting set problem to the problem of deciding whether there is a minimal always eventually configuration of size less than or equal to α. In hitting set problem n sets $A_1, A_2, ..., A_n$ are given where each set A_i for $1 \leq i \leq n$ is a subset of a universal set $\mathcal{A}$, and $\mathcal{A} = \cup_i A_i$. The problem of deciding whether there is a minimal set $B \subseteq \mathcal{A}$ with size less than or equal to some $0 < \beta \leq |\mathcal{A}|$ whose intersection with all A_i sets is not empty, i.e., $\forall 1 \leq i \leq n\ B \cap A_i \neq \emptyset$, is NP-hard.

Given the universal set $\mathcal{A}$, the sets A_i for $1 \leq i \leq n$, and the size β, we construct an LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$ such that there is a minimal hitting set B with size less than or equal to β if and only if there is a minimal always eventually configuration with size less than or equal to β. Assume an order over elements of $\mathcal{A} = \{a_1, a_2, ..., a_k\}$. Assume the elements of each set A_i are sorted with the same order. For any $a_i \in \mathcal{A}$ we consider a state $q_{a_i} \in Q$ for $\mathcal{T}$. We also consider two other states: $q_0 \in Q$ as the initial state, and $q_{sink} \in Q$ as an state with a transition to every state corresponding to the first element of each set. For each set $A_i = \{a_{i_1}, a_{i_2}, ..., a_{i_j}\}$, we add a transition $\delta(q_{a_{i_p}}) = q_{a_{i_{p+1}}}$ between states $q_{a_{i_p}}$ and $q_{a_{i_{p+1}}}$ for $1 \leq p < j$ corresponding to the consecutive elements a_{i_p} and $a_{i_{p+1}}$ of A_i. We connect the initial state q_0 to the state $q_{a_{i_1}}$ corresponding to the first element of A_i, and we connect the state $q_{a_{i_j}}$ corresponding to the last element of A_i to the state q_{sink}. All the runs of $\mathcal{T}$ eventually reaches q_{sink} which is connected to the states corresponding to first elements of sets A_i, that is, $\delta(q_{sink}) = \{q_{a_{1_1}}, q_{a_{2_1}}, ..., q_{a_{n_1}}\}$ where $q_{a_{i_1}}$ for $1 \leq i \leq n$ is the first element of the set A_i. The label function $\mathcal{L}$ is not important for the proof and we set it to empty for all states. It is easy to see that the construction can be done in polynomial time.

Intuitively, each run of $\mathcal{T}$ starts at q_0, non-deterministically chooses a set A_i, and visits the states $\mathcal{C}_{A_i} = \left\{q_{u_{i_1}}, q_{u_{i_2}}, ..., q_{u_{i_j}}\right\}$ corresponding to its elements in order and ends up at the state q_{sink}, where a set is chosen non-deterministically again, and the run continues. If B is a hitting set, removing its corresponding states from $\mathcal{T}$ will leave no cycle in $\mathcal{T}$ since q_{sink} is not reachable anymore. Thus, the configuration $\mathcal{C}_B$ corresponding to B is an always eventually configuration. Conversely, if $\mathcal{C}_B$ is an always eventually configuration, its corresponding set $B \subseteq \mathcal{A}$ is a hitting set. Therefore, any minimal always eventually configuration $\mathcal{C}_B$ corresponds to a minimal hitting set $B \subseteq \mathcal{A}$ and vice versa. $\square$

Algorithm 3.2: Synthesizing $\square\lozenge\psi_{\mathcal{P}}$ patterns

Input: LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$
Output: Patterns of the form $\square\lozenge\psi_{\mathcal{P}}$ where $\mathcal{T} \models \square\lozenge\psi_{\mathcal{P}}$

```
1  Patterns:=Empty;
2  visitedConfs:=⟨q_0⟩;
3  last := 1;
4  j := −1;
5  while true do
6      Next = {q_i ∈ Q | ∃q ∈ visitedConfs[last] ∧ ∃(q, q_i) ∈ δ};
7      if ∃k : visitedConf[k] = Next then
8          j := k;
9          break;
10     else
11         last := last + 1;
12         visitedConf[last] := Next;
13 foreach j ≤ i ≤ last do
14     Let Ψ_i := □◇ ⋁_{q∈visitedConf[i]} q;
15     Add Ψ_i to Patterns;
16 return Patterns;
```

Consequently, computing all minimal always-eventually patterns is infeasible in practice even for medium sized specifications. We propose an alternative algorithm that computes *some* of the always eventually patterns. Although the algorithm has an exponential upper-bound, it is simpler and terminated faster in our experiments, as it avoids enumerating all configurations. Algorithm 3.2 computes $\square\lozenge\psi_{\mathcal{P}}$ patterns, where $\psi_{\mathcal{P}}$ is a disjunction of a subset of states of $\mathcal{T}$. It starts with the configuration $\{q_0\}$, and at each step computes the next configuration, i.e., the set of states that the runs of $\mathcal{T}$ can reach at the next step from the current configuration. A sequence $C_0, C_1, ..., C_j$ of configurations is discovered during the search, where $C_0 = \{q_0\}$ and $j \geq 0$. The procedure terminates when a configuration C_i is reached that is already visited, i.e., there exists $0 \leq j < i$ such that $C_j = C_i$. There is a cycle between C_j and C_{i-1} and thus, all the configurations in the cycle will always eventually be visited over all runs of $\mathcal{T}$. In Algorithm 3.2 the set *visitedConfs* keeps track of the visited configurations, *last* is the index of the last configuration visited in *visitedConf*, and j is an index pointing to the configuration which is the start of the cycle. Initially, j is set to -1. Since there are only finite number of configurations, Algorithm 3.2 terminates, and it is of complexity $O(2^{|Q|})$.

Algorithm 3.3: Synthesizing $\Box\psi_{\mathcal{P}}$ pattern

Input: LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$
Output: Pattern of the form $\Box\psi_{\mathcal{P}}$ where $\mathcal{T} \models \Box\psi_{\mathcal{P}}$

1 Let $\psi_{\mathcal{P}} = \bigvee_{q \in Q} q$;
2 return $\Box\psi_{\mathcal{P}}$;

Synthesizing $\Box\psi_{\mathcal{P}}$ Pattern

For a given LTS $\mathcal{T}$, a safety pattern of the form $\Box\psi_{\mathcal{P}}$ is synthesized where $\psi_{\mathcal{P}}$ is simply the disjunction of all the states in $\mathcal{T}$, i.e., $\psi_{\mathcal{P}} = \bigvee_{q \in Q} q$. It is easy to see that removing any state from ψ leads to a formula that is not satisfied by $\mathcal{T}$ anymore. The synthesis procedure is of complexity $O(|Q|)$. Algorithm 3.3 computes a pattern of the form $\Box\psi_{\mathcal{P}}$. The following theorem states that computed $\Box\psi_{\mathcal{P}}$ pattern is the strongest formula of its specific form that holds over all runs of the input LTS.

Theorem 3.3. *The pattern $\Box\psi_{\mathcal{P}} = \Box \bigvee_{q \in Q} q$ is the strongest safety pattern that is satisfied over all runs of $\mathcal{T}$.*

Proof. First we show that any propositional formula over the states Q of $\mathcal{T}$ that holds over some run of it can be transformed into an equivalent formula in disjunctive form without negations.

Lemma 3.1. *Let $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$ be a labeled transition system. Consider a propositional formula ϕ over the states Q. Assume there exists a run σ of $\mathcal{T}$ and $i \geq 0$ such that $\sigma_i \models \phi$. Then there exists $Q_{\phi'} \subseteq Q$ such that the formulas ϕ and $\phi' = \bigvee_{q \in Q_{\phi'}} q$ are equivalent.*

Proof. Without loss of generality assume that ϕ only includes negation and disjunction connectives. Note that for any run σ of $\mathcal{T}$, $\sigma_i \models \neg \bigvee_{q \in Q'} q$ for some $Q' \subseteq Q$ and $i \geq 0$ if and only if $\sigma_i \models \bigvee_{q \in Q \setminus Q'} q$. Therefore, subformulas of the form $\neg \bigvee_{q \in Q'} q$ can be replaced by $\bigvee_{q \in Q \setminus Q'} q$. Using this rule, all negation operators can be removed from ϕ to obtain an equivalent formula ϕ'. Let $Q_{\phi'} \subseteq Q$ be the set of states $q \in Q$ which appears in ϕ'. It follows that the formulas ϕ and $\phi' = \bigvee_{q \in Q_{\phi'}} q$ are equivalent. □

From Lemma 3.1 it follows that any propositional formula over the states of $\mathcal{T}$ can be transformed to an equivalent formula in disjunctive form without negations. Obviously the safety pattern returned by the Algorithm 3.3 holds over all runs of $\mathcal{T}$. Removing any state from the computed pattern leads to a formula that is not satisfied by $\mathcal{T}$ anymore.

Algorithm 3.4: Synthesizing $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$ patterns

Input: LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$
Output: Patterns of the form $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$ where $\mathcal{T} \models \Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$

```
Let Patterns = Empty;
foreach state q ∈ Q do
    Patterns = Patterns ∪ □(q → ○ ⋁_{q'∈Next(q)} q');
return Patterns;
```

Therefore, the synthesized safety pattern is the strongest formula of the form $\Box\psi_{\mathcal{P}}$ such that $\mathcal{T} \models \Box\psi_{\mathcal{P}}$. □

Synthesizing $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$ Patterns

For a given LTS $\mathcal{T}$, a set of transition patterns of the form $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$ is synthesized as follows. Each $\psi_{\mathcal{P}}$ consists of a single state $q \in Q$, for which the $\psi'_{\mathcal{P}}$ is disjunction of its successors, i.e. $\psi'_{\mathcal{P}} = \bigvee_{q' \in Next(q)} q'$ where $Next(q) = \{q' \in Q \mid \delta(q) = q'\}$. Intuitively, each transition pattern states that, always when a state is visited, its successors will be visited at the next step. The synthesis procedure is of complexity $O(|Q| + |\delta|)$. Algorithm 3.4 summarizes the steps for computing patterns of the form $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$. The following theorem states that computed $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$ patterns are the strongest formulas of their specific form that hold over all runs of the input LTS.

Theorem 3.4. *Synthesized $\Box(\psi_{\mathcal{P}} \rightarrow \bigcirc\psi'_{\mathcal{P}})$ patterns are the strongest transition patterns that are satisfied over all runs of $\mathcal{T}$.*

Proof. Similar to proof of Theorem 3.3. □

Synthesizing $\Diamond\psi_{\mathcal{P}}$ Patterns

For an LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$, we say a configuration C is an *eventually configuration* if for any run $\sigma = \sigma_0\sigma_1\sigma_2\cdots$ of $\mathcal{T}$ there exists a state $q \in C$ and a time-step $i \geq 0$ such that $\sigma_i = q$. That is, any run of $\mathcal{T}$ eventually visits a state from configuration C. It follows that if C is an eventually configuration for $\mathcal{T}$, then $\mathcal{T} \models \Diamond \bigvee_{q \in C} q$. We say an eventually configuration C is *minimal* if there exists no $C' \subset C$ such that C' is an eventually configuration. Note that removing any state $q \in C$ from a minimal eventually configuration leads to a configuration which is not an eventually configuration.

Algorithm 3.5 constructs eventually patterns that correspond to minimal eventually configurations of $\mathcal{T}$ with size less than or equal to β which is specified by the user. The larger configurations usually lead to larger formulas that are harder for the user to understand. The user can specify the value of β. Heuristics can also be used to automatically set β based on the properties of $\mathcal{T}_c$, e.g., the maximum outdegree of the vertices in the corresponding directed graph $\mathcal{G}_{\mathcal{T}_c}$, where the outdegree of a vertex is the number of its outgoing edges. In Algorithm 3.5, the set $\Diamond$Configurations keeps the minimal eventually configurations discovered so far. Algorithm 3.5 initializes the sets Patterns and $\Diamond$Configurations to $\{\Diamond q_0\}$ and $\{q_0\}$, respectively. Note that $\Diamond q_0$ holds over all runs of $\mathcal{T}$ simply because all runs of $\mathcal{T}$ start from the initial state q_0. Algorithm 3.5 then checks each possible configuration $Q' \subseteq Q \setminus \{q_0\}$ with size less than or equal to β in a non-decreasing order of $|Q'|$ to find minimal eventually configurations. Without loss of generality, we assume that all states in $\mathcal{T}_c$ have outgoing edges[3]. At each iteration, a configuration Q' is chosen. It is then checked if there is a minimal eventually configuration Q'' that is already discovered and $Q'' \subset Q'$. If such Q'' exists, Q' is not minimal. Otherwise, the algorithm checks if it is an eventually configuration by first removing all the states in Q' and their corresponding incoming and outgoing transitions from $\mathcal{T}$ to obtain another LTS $\mathcal{T}'$. Now, if there is an infinite run from q_0 in $\mathcal{T}'$, then there is a run in $\mathcal{T}$ that does not visit any state in Q'. Otherwise, Q' is a minimal eventually configuration and is added to $\Diamond$Configurations. The corresponding formula $\Psi = \Diamond \bigvee_{q \in Q'} q$ is also added to the set of eventually patterns. Note that checking if there exists an infinite run in $\mathcal{T}'$ can be done by considering $\mathcal{T}'$ as a graph and checking if there is a reachable cycle from q_0, which can be done in linear time in number of states and transitions of $\mathcal{T}$. Therefore, the algorithm is of complexity $O(|Q|^{\beta}(|Q| + |\delta|))$. Computing all minimal eventually patterns is NP-hard, as stated in the following theorem[4].

Theorem 3.5. *Computing all minimal eventually configurations is NP-hard*

Proof. The proof is similar to that of Theorem 3.2. Given the universal set $\mathcal{A}$, the sets A_i for $1 \leq i \leq n$, and the size β, we construct an LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$ such that there is a minimal hitting set B with size less than or equal to β if and only if there is a minimal eventually configuration with size less than or equal to β other than the trivial

[3]A transition from any state with no outgoing transition can be added to a dummy state with a self loop. Patterns which include the dummy state will be removed.

[4]In practice, we use an idea similar to the one used in Algorithm 3.2 for computing *some* of the eventually patterns.

Algorithm 3.5: Synthesizing $\Diamond\psi_{\mathcal{P}}$ patterns

Input: LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$
Input: β: maximum number of states in generated patterns
Output: a set of patterns of the form $\Diamond\psi_{\mathcal{P}}$ where $\mathcal{T} \models \Diamond\psi_{\mathcal{P}}$

```
Patterns := {◇q0};
◇Configurations := {q0};
foreach Q' ⊆ Q\{q0} with non-decreasing order of |Q'| where |Q'| ≤ β do
    if ∄Q'' ∈ ◇Configurations s.t. Q'' ⊆ Q' then
        Let L'(q) = L(q) for all q ∈ Q\Q';
        Let δ' = {(q, q') ∈ δ | q ∉ Q' ∧ q' ∉ Q'};
        Let T' = ⟨Q\Q', {q0}, δ', L'⟩;
        if there is no infinite run from q0 in T' then
            Add Q' to ◇Configurations;
            Let Ψ = ◇ ⋁_{qi∈Q'} qi;
            Add Ψ to Patterns;
return Patterns;
```

eventually configuration $\{q_0\}$. LTS $\mathcal{T}$ is constructed similarly, and the only difference is the transitions from q_{sink}. We let $\delta(q_{sink}) = \{q_{sink}\}$, i.e., q_{sink} has a self-loop. Intuitively, each run of $\mathcal{T}$ starts at q_0, non-deterministically chooses a set A_i, and visits the states $\mathcal{C}_{A_i} = \left\{q_{u_{i_1}}, q_{u_{i_2}}, ..., q_{u_{i_j}}\right\}$ corresponding to its elements in order and ends up at the state q_{sink} and stays there forever. If B is a hitting set, removing its corresponding states from $\mathcal{T}$ will leave no infinite run in $\mathcal{T}$ since q_{sink} is not reachable anymore. Thus, the configuration $\mathcal{C}_B$ corresponding to B is an eventually configuration. Conversely, if $\mathcal{C}_B$ is an eventually configuration, its corresponding set $B \subseteq \mathcal{A}$ is a hitting set. Therefore, any minimal eventually configuration $\mathcal{C}_B \neq \{q_0\}$ corresponds to a minimal hitting set $B \subseteq \mathcal{A}$ and vice versa. □

Example 3.2. *Consider the LTS shown in Figure 3.3. Algorithm 3.5 starts at initial configuration* $\{q_0\}$ *and generates the formula* $\Diamond q_0$*. None of* $\{q_1\}$*,* $\{q_2\}$ *or* $\{q_3\}$ *is an eventually configuration. For example for configuration* $\{q_1\}$*, there exists the run* $\sigma = q_0, (q_3)^\omega$ *which never visits* q_1*. Configurations* $\{q_1, q_3\}$ *and* $\{q_2, q_3\}$ *are minimal eventually configurations. For example removing* $\{q_1, q_3\}$ *will lead to an LTS with no infinite run (no cycle is reachable from* q_0 *in the corresponding graph). It is easy to see that configuration* $\{q_1, q_2\}$ *is not an eventually configuration. Configuration* $\{q_1, q_2, q_3\}$ *is not minimal, although it is an eventually configuration. Thus Algorithm 3.5 returns the set of patterns* $\{\Diamond q_0, \Diamond(q_1 \vee q_3), \Diamond(q_2 \vee q_3)\}$*.*

Algorithm 3.6: Synthesizing $\Diamond\Box\psi_{\mathcal{P}}$ patterns

Input: LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$

Output: A set of patterns of the form $\Diamond\Box\psi_{\mathcal{P}}$ where $\mathcal{T} \models \Diamond\Box\psi_{\mathcal{P}}$

1 Let $Q^{cycle} = \{q \in Q \mid \text{there exists a cycle } \in \mathcal{T} \text{ including } q\}$;

2 return $\Psi = \Diamond\Box \bigvee_{q \in Q^{cycle}} q$;

Synthesizing $\Diamond\Box\psi_{\mathcal{P}}$ Patterns

To compute formulas of the form $\Diamond\Box\psi_{\mathcal{P}}$ that hold over all runs of an LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$, we view $\mathcal{T}$ as a graph and partition its states into two groups: $Q^{cycle} \subseteq Q$, the set of states that are part of a cycle in $\mathcal{T}$ (including the cycle from one node to itself), and $Q' = Q \backslash Q^{cycle}$. Without loss of generality we assume that any state $q \in Q$ is reachable from q_0. Therefore, any state $q \in Q^{cycle}$ belongs to a reachable strongly connected component SCC of $\mathcal{T}$. Also for any strongly connected component SCC of $\mathcal{T}$, there exists a run σ of $\mathcal{T}$ that reaches states in SCC and keeps cycling there forever. Hence, the formula $\Psi_1 = \Diamond\Box \bigvee_{q \in SCC} q$ holds over the run σ. Indeed Ψ_1 is the minimal formula of disjunctive form that holds over all runs that can reach the strongly connected component SCC. That is, by removing any of the states from Ψ_1, one can find a run σ' that can reach the strongly connected component SCC and visit the removed state, falsifying the resulted formula. Therefore, eventually for any execution of $\mathcal{T}$, the state of the system will always be in one of the states $q \in Q^{cycle}$. Thus the formula $\Psi = \Diamond\Box \bigvee_{q \in Q^{cycle}} q$ is the minimal formula of the form $\Diamond\Box\psi_{\mathcal{P}}$ that holds over all runs of $\mathcal{T}$. Algorithm 3.6 summarizes the steps for synthesizing the patterns of the form $\Diamond\Box\psi_{\mathcal{P}}$. To partition the states of the $\mathcal{T}$ into Q^{cycle} and Q', we use Tarjan's algorithm for computing strongly connected components of the graph. Thus the algorithm is of linear time complexity in number of states and transitions of $\mathcal{T}$.

Example 3.3. *Consider the LTS shown in Figure 3.3. It has three strongly connected components:* $\{q_0\}$, $\{q_1, q_2\}$ *and* $\{q_3\}$. *Only the latter two components include a cycle inside them, that is* $Q^{cycle} = \{q_1, q_2, q_3\}$. *Thus, the pattern* $\Psi = \Diamond\Box(q_1 \vee q_2 \vee q_3)$ *is generated. Note that the possible runs of the system are* $\sigma_1 = q_0, (q_1, q_2)^\omega$ *and* $\sigma_2 = q_0, (q_3)^\omega$. *The generated pattern* Ψ *holds over both of these runs. Observe that removing any of the states in* Ψ *will result in a formula that is not satisfied by* $\mathcal{T}$ *any more.*

Synthesizing $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$ Patterns

To generate candidates of the form $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$, first note that $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$ holds only if $\Diamond\psi_{\mathcal{P}}$ holds. Therefore, a set of eventually patterns $\Diamond\psi_{\mathcal{P}}$ is first computed using Algorithm 3.5. Then for each formula $\Diamond\psi_{\mathcal{P}}$, the pattern $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc \bigvee_{q \in Next(\psi_{\mathcal{P}})} q)$ is generated, where $Next(\psi_{\mathcal{P}})$ is the set of states that can be reached in one step from the configuration specified by $\psi_{\mathcal{P}}$. Formally, $Next(\psi_{\mathcal{P}}) = \{q_i \in Q \mid \exists q_j \in \mathcal{C} \text{ s.t. } (q_j, q_i) \in \delta\}$ and $\mathcal{C}$ is the configuration represented by $\psi_{\mathcal{P}}$, i.e., $\psi_{\mathcal{P}} = \bigvee_{q \in \mathcal{C}} q$. The most expensive part of this procedure is computing the eventually patterns, therefore its complexity is the same as Algorithm 3.5. Algorithm 3.7 summarizes the steps for synthesizing patterns of the form $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$.

Algorithm 3.7: Synthesizing $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$ patterns

Input: LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$
Input: β, maximum number of states in eventually configurations
Output: a set of patterns of the form $\Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$ where $\mathcal{T} \models \Diamond(\psi_{\mathcal{P}} \wedge \bigcirc\psi'_{\mathcal{P}})$

```
1 ◇Patterns = eventually patterns generated by Algorithm 3.5 with input T and β;
2 Patterns := Empty;
3 foreach formula ◇ψ_P ∈ ◇Patterns do
4     Patterns = Patterns ∪ { ◇(ψ_P ∧ ○ ⋁_{q∈Next(ψ_P)} q) };
5 return Patterns;
```

Example 3.4. *Consider the LTS shown in Figure 3.3. Given the eventually formulas* $\Diamond q_0$, $\Diamond(q_1 \vee q_3)$, *and* $\Diamond(q_2 \vee q_3)$ *produced in Example 3.2, patterns* $\Diamond(q_0 \wedge \bigcirc(q_1 \vee q_3))$, $\Diamond((q_1 \vee q_3) \wedge \bigcirc(q_2 \vee q_3))$ *and* $\Diamond((q_2 \vee q_3) \wedge \bigcirc(q_1 \vee q_3))$ *are generated.*

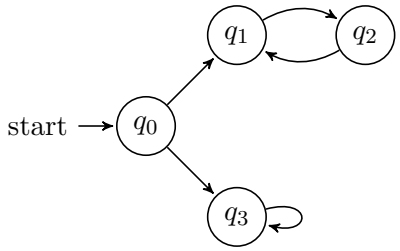


Figure 3.3: An LTS $\mathcal{T}$

The following theorem states that the procedures described above generate the strongest patterns of the specified forms.

Theorem 3.6. *For any formula of the form $\Diamond\psi, \Diamond\Box\psi$, or $\Diamond(\psi_1 \wedge \bigcirc\psi_2)$ that holds over all runs of a given LTS $\mathcal{T}$, there is an equivalent or stronger formula of the same form synthesized by the algorithms 3.5, 3.6 and 3.7, respectively.*

Proof. Note that if C is an eventually configuration, then any configuration C' such that $C \subset C'$ is also an eventually configuration. Moreover, $\Diamond \bigvee_{q\in C} q \rightarrow \Diamond \bigvee_{q'\in C'} q'$, that is, the formula corresponding to C is stronger than the one corresponding to C'. We prove the theorem for Algorithms 3.5 and 3.6. The proof for Algorithm 3.7 is similar. First consider the eventually formulas $\Diamond\psi_{\mathcal{P}}$ generated by Algorithm 3.5. We assume that $\beta = 2^{|Q|}$, that is, the algorithm finds all minimal eventually configurations. Assume there exists a formula $\Diamond\phi$ which holds over all runs of $\mathcal{T}$. By Lemma 3.1 there exists $Q_\phi \subseteq Q$ such that $\Diamond\phi = \Diamond(\bigvee_{q\in Q_\phi} q)$. Since $\Diamond\phi$ holds over all runs of $\mathcal{T}$, Q_ϕ must be an eventually configuration. Algorithm 3.5 finds all minimal eventually configurations of $\mathcal{T}$. Therefore, there exists a minimal eventually configuration $Q_\psi \subseteq Q$ corresponding to a formula $\Diamond\psi$ generated by Algorithm 3.5 such that $Q_\psi \subseteq Q_\phi$. It follows that $\Diamond\psi \rightarrow \Diamond\phi$. That is, there exists a formula generated by Algorithm 3.5 which is stronger than or equivalent to $\Diamond\phi$.

Eventually always formula $\Diamond\Box\psi$ generated by Algorithm 3.6 is such that removing any state from ψ makes the formula unsatisfiable and adding any state to it makes the formula weaker. Thus any formula $\Diamond\Box\phi$ which holds over all runs of $\mathcal{T}$ should be equivalent to or weaker than $\Diamond\Box\psi$. □

3.2.3 Instantiating Patterns

To obtain LTL formulas over a specified subset U of variables from patterns, we replace the states in patterns by their projected labels. For example, from an eventually pattern $\Diamond\psi_{\mathcal{P}} = \Diamond(\bigvee_{q\in Q_{\psi_{\mathcal{P}}}} q)$ where $Q_{\psi_{\mathcal{P}}} \subseteq Q$ is a configuration for $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$, we obtain the formula $\psi = \Diamond(\bigvee_{q\in Q_{\psi_{\mathcal{P}}}} \mathcal{L}(q)_{|U})$.

Example 3.5. *Let $\Sigma = \{a, b, c\}$ be the set of variables. Consider the LTS $\mathcal{T}$ shown in Figure 3.4, where $\mathcal{L}(q_0) = \neg a \wedge \neg b \wedge \neg c$, $\mathcal{L}(q_1) = \neg a \wedge b \wedge \neg c$, $\mathcal{L}(q_2) = a \wedge \neg b \wedge \neg c$, $\mathcal{L}(q_3) = \neg a \wedge b \wedge \neg c$ Let $U = \{a, b\}$ be the set of variables specified by the designer to be used in all forms of formulas. Figure 3.5 shows $\mathcal{T}^a$ which is an abstraction of $\mathcal{T}$ with respect to U, where the mapping function F is defined such that $F^{-1}(q_0^a) = \{q_0\}$, $F^{-1}(q_1^a) = \{q_1, q_3\}$, and $F^{-1}(q_2^a) = \{q_2\}$, and the labels are defined as $\mathcal{L}^a(q_0^a) = \neg a \wedge \neg b$, $\mathcal{L}^a(q_1^a) = \neg a \wedge b$, and $\mathcal{L}^a(q_2^a) = a \wedge \neg b$. A set of patterns are synthesized using the input LTS. For example,*

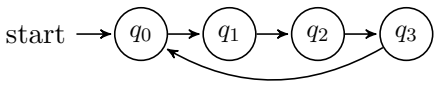


Figure 3.4: An LTS $\mathcal{T}$

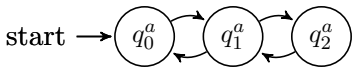


Figure 3.5: Abstract LTS $\mathcal{T}^a$ of $\mathcal{T}$

$\psi_{\mathcal{P}} = \Diamond(q_1^a)$ is an eventually pattern where $\mathcal{T}^a \models \psi_{\mathcal{P}}$, meaning that eventually over all runs of the $\mathcal{T}^a$ the state q_1^a is visited. An LTL formula is obtained using the patterns, labels and specified subset of variables. For example, $\Psi = \Diamond(\neg a \wedge b)$ is obtained from the pattern $\psi_{\mathcal{P}}$, where the state q_1^a is replaced by its label. Note that the formula $\Psi' = \Diamond((\neg a \wedge b) \wedge \bigcirc(a \wedge \neg b))$ can be synthesized from the pattern $\Psi'_{\mathcal{P}} = \Diamond(q_1 \wedge \bigcirc q_2)$ from $\mathcal{T}$, however, $\mathcal{T}^a$ does not satisfy Ψ'. A more conservative formula $\Diamond((\neg a \wedge b) \wedge \bigcirc((a \wedge \neg b) \vee (\neg a \wedge \neg b))$ is obtained using the abstraction.

Remark 3.1. *Patterns can be synthesized from either $\mathcal{T}$ or $\mathcal{T}^a$. It is sometimes necessary to use $\mathcal{T}^a$ due to the high complexity of the algorithms for computing certain types of patterns (e.g., eventually patterns), as $\mathcal{T}^a$ may have significantly less number of states compared to $\mathcal{T}$ which improves the scalability of the methods. However, abstraction may introduce additional non-determinism into the model, leading to refinements that are more "conservative." Besides, some of the formulas which are satisfied by $\mathcal{T}$, cannot be computed from $\mathcal{T}^a$. It is up to the user to choose techniques that serve her purposes better.*

3.3 Counter-Strategy-Guided Refinement of Unrealizable GR(1) Specifications

Writing a correct and complete formal specification that conforms to the (informal) design intent is a hard and tedious task [CHJ08, KHB09]. Initial specifications are often incomplete and unrealizable. Unrealizability of the specification is often due to inadequate environment assumptions. In other words, assumptions about the environment are too weak, leading to an environment with too many possible behaviors that make it impossible for the system to satisfy the specification. Usually there is only a rough and incomplete model of the environment in the design phase; thus it is easy to miss assumptions on the environment side. We would like to automatically find such *missing* assumptions that can be added to the specification and make it realizable. Computed assumptions can be used to give the user insight into the specification. They also provide ways to correct the specification. In

the context of compositional synthesis [KPV06, OTM11], derived assumptions based on the components specifications can be used to construct interface rules between the components.

An unrealizable specification cannot be executed or simulated which makes its debugging a challenging task. Counter-strategies are used to explain the reason for unrealizabilty of LTL specifications [KHB09]. Intuitively, a counter-strategy defines how the environment can react to the outputs of the system in order to enforce the system to violate the specification. Konighofer et al. in [KHB09] show how such a counter-strategy can be computed for an unrealizable LTL specification. The requirement analysis tool RATSY [BCG$^+$10] implements their method for GR(1) specifications.

Counter-strategies can still be difficult to understand by the user especially for larger systems. In this section, we propose a debugging approach that uses the counter-strategies to strengthen the assumptions on the environment in order to make the specification realizable. For a given unrealizable specification, our algorithm analyzes the counter-strategy and synthesizes a set of *candidate* assumptions in the forms that are allowed in GR(1) specifications. Any of the computed candidate assumptions, if added to the specification, restricts the environment in such a way that it cannot behave according to the counter-strategy—without violating its assumptions—anymore. Thus we say the counter-strategy is *ruled out* from the environment's possible behaviors by adding the candidate assumption to the specification. We now formally define the problem considered in the rest of this section.

Problem Statement 3.2. *Given a GR(1) specification $\Phi = \Phi_e \rightarrow \Phi_s$ that is satisfiable but unrealizable, find a refinement $\Psi = \bigwedge_i \Psi_i$ as a conjunction of environment assumptions Ψ_i such that $\Phi_e \wedge \Psi$ is satisfiable and $\Phi_e \wedge \Psi \rightarrow \Phi_s$ is realizable.*

Specification refinements are constructed in two phases. First, given a counter-strategy's Moore machine M_c, we build an abstraction which is an LTS $\mathcal{T}$. A set of patterns are then synthesized over $\mathcal{T}$. These patterns along with a subset of variables specified by the user are used to generate a set of LTL formulas that hold over *all* runs of M_c. We ask the user to specify a subset of variables which she thinks contribute to the unrealizability of the specification. This set can also be used to guide the algorithm to generate formulas over the set of variables which are underspecified. Using a smaller subset of variables leads to simpler formulas that are easier for the user to understand.

The complement of the generated formulas form the set of candidate assumptions that

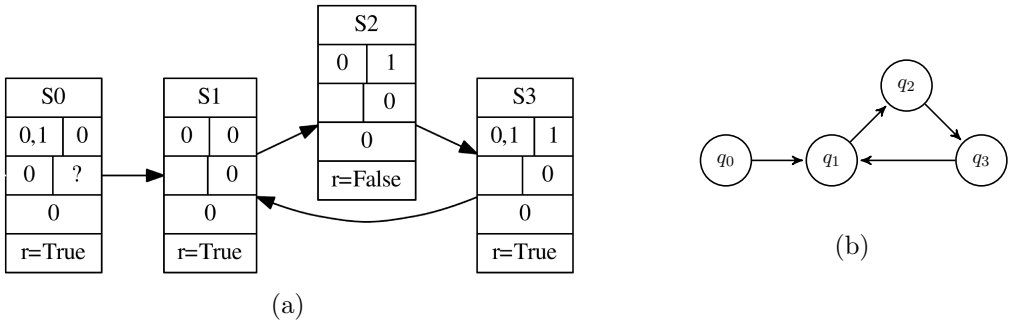


Figure 3.6: (a) A counter-strategy produced by RATSY for the specification of Example 3.6 with the additional assumption $\square\lozenge(\neg r)$, where $c = \texttt{true}$ holds in all states. (b) The LTS $\mathcal{T}$ corresponding to the counter-strategy of part (a).

can be used to rule out the counter-strategy from the environment's possible behaviors. We remove the candidates that are not consistent with the specification in order to avoid a trivial solution `false`. Note that adding inconsistent assumptions leads to an unsatisfiable environment, i.e. $\Phi_e = \texttt{false}$ and the specification $\Phi_e \rightarrow \Phi_s$ is trivially realizable, but obviously it is not an interesting solution.

Any assumption from the set of generated candidates can be used to rule out the counter-strategy. Our approach does a breadth-first search over the candidates. If adding any of the candidates makes the specification realizable, the algorithm returns that candidate as a solution. Otherwise at each iteration, the process is repeated for any of the new specifications resulting from adding a candidate. The depth of the search is controlled by the user. The search continues until either a consistent refinement is found or the algorithm cannot find one within the specified depth (hence the search algorithm is sound, but not complete).

Example 3.6. *Let $\mathcal{I} = \{r, c\}$ and $\mathcal{O} = \{g, v\}$ be the set of input and output variables, respectively. Here r, c, g and v stand for* request, clear, grant *and* valid *signals, respectively. We start with no assumption, that is, we assume $\Phi_e = \texttt{true}$. Consider the following system guarantees:*

- $\Phi_1 = \square(r \rightarrow \bigcirc\lozenge g)$
- $\Phi_2 = \square((c \vee g) \rightarrow \bigcirc\neg g)$
- $\Phi_3 = \square(c \rightarrow \neg v)$
- $\Phi_4 = \square\lozenge(g \wedge v)$

Let $\Phi_s = \bigwedge_{i=1}^{4} \Phi_i$ be the conjunction of these formulas. Φ_1 requires that every request must be granted eventually starting from the next step by setting signal g to high. Φ_2 says that

if clear or grant signal is high, then grant must be low at the next step. Φ_3 *says if clear is high, then the valid signal must be low. Finally,* Φ_4 *says that system must issue a valid grant infinitely often.*

The specification $\Phi_e \rightarrow \Phi_s$ *is unrealizable. A simple counter-strategy is for the environment to keep* r *and* c *high at all times. Then, by* Φ_3*,* v *needs to be always low and thus* Φ_4 *cannot be satisfied by any system. RATSY produces this counter-strategy which is then fed to our algorithm. An example candidate synthesized to rule out this counter-strategy is the assumption* $\Psi = \square\Diamond(\neg r)$*. Adding* Ψ *to the specification prevents the environment from always keeping* r *high, thus the environment cannot use the counter-strategy anymore. However, the specification* $\Phi_e \wedge \Psi \rightarrow \Phi_s$ *is still unrealizable. RATSY produces the counter-strategy shown in Figure 3.6(a) for the new specification. The new counter-strategy keeps the* c *high all the times. The value of* r *is changed depending on the state of the counter-strategy as shown in Figure 3.6(a). The top block in each state of Figure 3.6(a) is the name of the state and the bottom block is the value(s) of the environment variables. Variables which have constant value over all states are given separately, for example signal* c *is always high for the counter-strategy in Figure 3.6. RATSY produces additional information, shown in middle blocks, on how the counter-strategy enforces the system to violate the specification. We do not use this information in the current version of the algorithm.*

The following formulas are examples of consistent refinements produced by our algorithm for the specification $\Phi_e \rightarrow \Phi_s$ *as possible ways to resolve the unrealizability of the specification:*

- $\Psi_1 = \square(\neg r \vee \neg c) \wedge \square(r \vee \neg c)$
- $\Psi_2 = \square(r \rightarrow \bigcirc\neg c) \wedge \square(\neg r \rightarrow \bigcirc\neg c)$
- $\Psi_3 = \square\Diamond(\neg r) \wedge \square(\neg c \vee r) \wedge \square(\neg r \rightarrow \bigcirc\neg c))$

Assumptions in both of the refinements Ψ_1 *and* Ψ_2 *imply* $\square(\neg c)$*, that is, adding them requires the environment to keep the signal* c *always low. Although adding these assumptions make the specification realizable, it may not conform to the design intent. Refinement* Ψ_3 *does not restrict* c *like* Ψ_1 *and* Ψ_2*, and only assumes that the environment sets the signal* r *to low infinitely often and that, when the request signal is low, the clear signal should be low at the same and the next step. Refinements* Ψ_1 *and* Ψ_2 *may also remind the designer to add the assumption* $\square\Diamond(c)$ *to the specification, meaning that clear happens infinitely often. By running the algorithm on this new specification, we only get* Ψ_3*, since both* Ψ_1 *and* Ψ_2 *are inconsistent with the new specification.*

Algorithm 3.8: Specification Refinement

Input: $\Phi = \Phi_e \rightarrow \Phi_s$, initial specification
Input: U, set of variables to be used in patterns
Input: α, maximum depth of the search
Output: Ψ, additional assumptions such that $\Phi_e \wedge \Psi \rightarrow \Phi_s$ is realizable

```
1  M_c := CounterStrategy(Φ);
2  CandidatesQ := GenerateCandidates(M_c,U);
3  while CandidatesQ is not Empty do
4      Ψ := CandidatesQ.DeQueue;
5      if Consistent(Φ,Ψ) then
6          Φ_new = Φ_e ∧ Ψ → Φ_s;
7          if Realizable(Φ_new) then
8              return Ψ;
9          else
10             if Depth(Ψ) < α then
11                 M_c := CounterStrategy(Φ_new);
12                 newCandidates := GenerateCandidates(M_c,U) ;
13                 foreach Ψ_new ∈ newCandidates do
14                     CandidatesQ.EnQueue(Ψ ∧ Ψ_new);
15 return No refinement was found;
```

3.3.1 Refining Unrealizable Specifications

Algorithm 3.8 finds environment assumptions that can be added to the specification to make it realizable. It gets as input the initial unrealizable specification $\Phi = \Phi_e \rightarrow \Phi_s$, the set U of variables to be used in generated assumptions and the maximum depth α of the search. It outputs a consistent refinement Ψ, if it can find one within the specified depth. The complements of these formulas form the set of candidate assumptions that can be used to rule out the counter-strategy. For an unrealizable specification, a counter-strategy is computed as a Moore transducer using the techniques proposed in [BCG$^+$10, KHB09]. Procedure **GenerateCandidates** produces a set of candidate assumptions in the forms allowed by GR(1) as follows: It first infers a set of formulas of the forms $\Diamond\Box\psi$, $\Diamond\psi$ and $\Diamond(\psi \wedge \bigcirc\psi')$ (complement of the forms allowed in GR(1)) using the methods described in Section 3.2. The complements of these formulas are of desired GR(1) forms and form the set of candidate assumptions that can be used to rule out the counter-strategy.

Algorithm 3.8 runs a breadth-first search to find a consistent refinement. Each node of

the search tree is a generated candidate assumption, while the root of the tree corresponds to the assumption `true` (i.e., no assumption). Each path of the search tree starting from the root corresponds to a candidate refinement as a conjunction of candidate assumptions of the nodes visited along the path. When a node is visited during the search, its corresponding candidate refinement is added to the specification. If the new specification is consistent and realizable, the refinement is returned by the algorithm. Otherwise, if the depth of the current node is less than the maximum specified, a set of candidate assumptions are generated based on the counter-strategy for the new specification and the search tree expands.

In Algorithm 3.8, the queue *CandidatesQ* keeps the candidate refinements that are found during the search. At each iteration, a candidate refinement Ψ is removed from the head of the queue. The procedure **Consistent** checks if Ψ is consistent with the specification Φ. If it is, the algorithm checks the realizability of the new specification $\Phi_{new} = \Phi_e \wedge \Psi \rightarrow \Phi_s$ using the procedure **Realizable** [BJP+12, BCG+10]. If Φ_{new} is realizable, Ψ is returned as a suggested refinement. Otherwise, if the depth of the search for reaching the candidate refinement Ψ is less than α, a new set of candidate assumptions are generated using the counter-strategy computed for Φ_{new}. Algorithm 3.8 keeps track of the number of counter-strategies produced along the path to reach a candidate refinement in order to compute its depth (**Depth(ψ)**). Each new candidate assumption Ψ_{new} results in a new candidate refinement $\Psi \wedge \Psi_{new}$ which is added to the end of the queue for future processing . The algorithm terminates when either a consistent refinement Ψ is found, or there is no more candidates in the queue to be processed.

Remark 3.2. *Note that there might be repetitive formulas among the generated candidates. We remove the repeated formulas in order to prevent the process from checking the same assumption repeatedly.*

3.3.2 Removing the Restrictive Formulas

Given two non-equivalent predicates ϕ_1 and ϕ_2, we say ϕ_1 is *stronger* than ϕ_2 if $\phi_1 \rightarrow \phi_2$ holds. Assume Ψ_1 and Ψ_2 are LTL formulas that hold over all runs of the counter-strategy computed for the specification $\Phi_e \rightarrow \Phi_s$, and that $\Psi_1 \rightarrow \Psi_2$. Note that $\neg\Psi_2 \rightarrow \neg\Psi_1$ also holds, i.e., $\neg\Psi_1$ is a *weaker* assumption compared to $\neg\Psi_2$. Adding either $\neg\Psi_1$ or $\neg\Psi_2$ to the environment assumptions Φ_e rules out the counter-strategy. However, adding the stronger assumption $\neg\Psi_2$ restricts the environment more than adding $\neg\Psi_1$. That is, $\Phi_e \wedge \neg\Psi_2$ puts

more constraints on the environment compared to $\Phi_e \wedge \neg\Psi_1$.

As an example, consider the counter-strategy M_c shown in Figure 3.6(a). Both $\Psi_1 = \Diamond(c \wedge \neg r)$ and $\Psi_2 = \Diamond(c)$ hold over all runs of M_c. Moreover, $\Psi_1 \rightarrow \Psi_2$. Consider the corresponding assumptions $\neg\Psi_1 = \Box(\neg c \vee r)$ and $\neg\Psi_2 = \Box(\neg c)$. Adding $\neg\Psi_2$ restricts the environment more than adding $\neg\Psi_1$. The refinement $\neg\Psi_2$ requires the environment to keep the signal c always low, whereas in case of $\neg\Psi_1$, the environment is free to assign additional values to its variables. It only prevents the environment from setting c to high and r to low at the same time.

We construct patterns that are strongest formulas of their specified form that hold over all runs of the counter-strategy. Removing the weaker patterns leads to shorter formulas that are easier for the user to understand. It also decreases the number of generated candidates at each step. More importantly, the generated candidate assumptions are the weakest formulas that can be constructed for the given structure and the user specified subset of variables. If the restriction imposed by any of these candidates is not enough to make the specification realizable, the method analyzes the counter-strategy computed for the new specification to find assumptions that can restrict the environment more. This way the counter-strategies guide the method to synthesize assumptions that can be used to achieve realizability.

3.3.3 Examples

We now illustrate the counter-strategy-guided refinement method with two examples. We use RATSY to generate counter-strategies and Cadence SMV model checker [McM] to check the consistency of the generated candidates. In our experiments, we set α in Algorithm 3.8 to 2, and β in Algorithm 3.5 to the maximum outdegree of the vertices of the counter-strategy's abstract directed graph. We slightly change Algorithm 3.8 to find all possible refinements within the specified depth.

Lift Controller

We borrow the lift controller example from [BJP$^+$12]. Consider a lift controller serving three floors. Assume that the lift has three buttons, denoted by the Boolean variables b_1, b_2 and b_3, which are controlled by the environment. The location of the lift is represented using Boolean variables f_1, f_2 and f_3 controlled by the system. The lift may be requested on each floor by pressing the corresponding button. We assume that (1) once a request is made, it cannot be

withdrawn, (2) once the request is fulfilled it is removed, and (3) initially there are no requests. Formally, the specification of the environment is $\Phi_e = \phi^e_{init} \wedge \Phi^e_{1_1} \wedge \Phi^e_{1_2} \wedge \Phi^e_{1_3} \wedge \Phi^e_{2_1} \wedge \Phi^e_{2_2} \wedge \Phi^e_{2_3}$, where

- $\phi^e_{init} = (\neg b_1 \wedge \neg b_2 \wedge \neg b_3)$,
- $\Phi^e_{1_i} = \Box(b_i \wedge f_i \rightarrow \bigcirc \neg b_i)$, and
- $\Phi^e_{2_i} = \Box(b_i \wedge \neg f_i \rightarrow \bigcirc b_i)$.

The lift initially starts on the first floor. We expect the lift to be only on one of the floors at each step. It can move at most one floor at each time step. We want the system to eventually fulfill all the requests. Formally the specification of the system is given as $\Phi_s = \phi^s_{init} \wedge \Phi^s_1 \bigwedge_i \Phi^s_{2,i} \wedge \Phi^s_3 \bigwedge_j \Phi^s_{4,j} \wedge \Phi^s_5$, where

- $\phi^s_{init} = f_1 \wedge \neg f_2 \wedge \neg f_3$,
- $\Phi^s_1 = \Box(\neg(f_1 \wedge f_2) \wedge \neg(f_2 \wedge f_3) \wedge \neg(f_1 \wedge f_3))$,
- $\Phi^s_{2,i} = \Box(f_i \rightarrow \bigcirc(f_{i-1} \vee f_i \vee f_{i+1}))$,
- $\Phi^s_3 = \Box((f_1 \wedge \bigcirc f_2) \vee (f_2 \wedge \bigcirc f_3) \rightarrow (b_1 \vee b_2 \vee b_3))$, and
- $\Phi^s_{4,j} = \Box\Diamond(b_j \rightarrow f_j)$.

The requirement Φ^s_3 says that the lift moves up one floor only if some button is pressed. The specification $\Phi = \Phi_e \rightarrow \Phi_s$ is realizable. Now assume that the designer wants to ensure that all floors are infinitely often visited; thus she adds the guarantees $\bigwedge_j \Phi^s_{5,j}$ where $\Phi^s_{5,j} = \Box\Diamond(f_j)$ to the set of system requirements. The specification $\Phi' = \Phi_e \rightarrow \Phi_s \bigwedge_j \Phi^s_{5,j}$ is not realizable. A counter-strategy for the environment is to always keep all b_i's low. We run our algorithms with the set of all the environment variables $\{b_1, b_2, b_3\}$ for all assumption forms. The algorithm generates the refinements $\Psi_1 = \Box\Diamond(b_1 \vee b_2 \vee b_3)$ and $\Psi_2 = \Box((\neg b_1 \wedge \neg b_2 \wedge \neg b_3) \rightarrow \bigcirc(b_1 \vee b_2 \vee b_3))$. Refinement Ψ_1 requires that the environment infinitely often presses a button. Refinement Ψ_2 is another suggestion which requires the environment to make a request after any inactive turn. Refinement Ψ_1 seems to be more reasonable and the user can add it to the specification to make it realizable.

Only one counter-strategy is processed during the search for finding refinements and three candidate assumptions are generated overall, where one of the candidates is inconsistent with Φ' and the two others are refinements Ψ_1 and Ψ_2. Thus, the search terminates after checking the generated assumptions at first level. Only 0.6 percent of total computation time was spent on generating candidate assumptions from the counter-strategy. Note that

to generate Ψ_1 using the template-based method in [LDS11], the user needs to specify a template with three variables which leads to $2^3 = 8$ candidate assumptions, although only one of them is satisfied by the counter-strategy.

AMBA AHB

ARM's Advanced Microcontroller Bus Architecture (AMBA) defines the Advanced High-Performance Bus (AHB) which is an on-chip communication protocol. Up to 16 *masters* and 16 *slaves* can be connected to the bus. The masters start the communication (read or write) with a slave and the slave responds to the request. Multiple masters can request the bus at the same time, but the bus can only be accessed by one master at a time. A bus access can be a single *transfer* or a *burst*, which consists of multiple number of transfers. A bus access can be locked, which means it cannot be interrupted. Access to the bus is controlled by the *arbiter*. More details of the protocol can be found in [BJP+12]. We use the specification given by one of RATSY's example files (amba02.rat). There are four environment signals:

- $\texttt{HBUSREQ}[i]$: Master i requests access to the bus.
- $\texttt{HLOCK}[i]$: Master i requests a locked access to the bus. This signal is raised in combination with $\texttt{HBUSREQ}[i]$.
- $\texttt{HBURST}[1:0]$: Type of transfer. It can be SINGLE (a single transfer), BURST4 (a four-transfer), or INCR (unspecified length burst).
- HREADY: Raised if the slave has finished processing the data. The bus owner can change and transfers can start only when HREADY is high.

The first three signals are controlled by the masters and the last one is controlled by the slaves. The specification of amba02.rat consists of one master and two slaves. For our experiment, we remove the fairness assumption $\square\lozenge\texttt{HREADY}$ from the specification. The new specification is unrealizable. We run our algorithm with the sets of variables $\{\texttt{HREADY}\}$, $\{\texttt{HREADY}, \texttt{HBUSREQ}[0], \texttt{HBUSREQ}[1], \texttt{HLOCK}[0], \texttt{HLOCK}[1]\}$, $\{\texttt{HREADY}\}$ and $\{\texttt{HBUSREQ}[0], \texttt{HBUSREQ}[1]\}$ to be used in liveness, safety, left and right hand side of transition assumptions, respectively. Some of the refinements generated by our method are:

- $\Psi_1 = \square\lozenge\texttt{HREADY}$,
- $\Psi_2 = \square(\texttt{HREADY} \vee \neg\texttt{HBUSREQ}[0] \vee \neg\texttt{HLOCK}[0] \vee \neg\texttt{HBUSREQ}[1] \vee \neg\texttt{HLOCK}[1]) \wedge \square\lozenge\texttt{HREADY}$, and
- $\Psi_3 = \square(\texttt{HREADY} \rightarrow \bigcirc\neg\texttt{HBUSREQ}[0]) \wedge \square(\neg\texttt{HREADY} \rightarrow \bigcirc\neg\texttt{HBUSREQ}[0])$.

Note that although Ψ_2 is a consistent refinement, it includes Ψ_1 as a subformula and it is more restrictive. The refinement Ψ_3 implies that `HBUSREQ[0]` must always be low from the second step on. Among these suggested refinements, Ψ_1 appears to be the best option. Our method only processed one counter-strategy with five states and generates five candidate assumptions to find the first refinement Ψ_1. To find all refinements within the depth two, overall five counter-strategies are processed by our method during the search, where the largest counter-strategy had 25 states. The number of assumptions generated for each counter-strategy during the search is less than nine. 28.6 percent of total computation time was spent on generating candidate assumptions from the counter-strategies.

3.4 Compositional Refinement

We propose three approaches for compositional refinement of the specifications Φ_1 and Φ_2 in the problem stated in Section 5.1. These approaches differ mainly in how much information about the strategy of the realizable component is shared with the unrealizable component. All three approaches use bounded search to compute the refinements. The search depth (number of times the refinement procedure can be called recursively) is specified by the user. Note that the proposed approaches are not complete, i.e., not finding a refinement does not mean that there is no refinement.

Approach 1 ("No knowledge of the strategy of C_1"): One way to synthesize the refinements Ψ and Ψ' is to compute a refinement Ψ' for the unrealizable specification Φ_2 using the counter-strategy-guided refinement method described in the previous section. The specification Φ_2 is refined by adding assumptions on its environment that rule out all the counter-strategies for Φ_2, and the refined specification $\Phi_2^{ref} = (\Phi_{e_2} \wedge \Psi') \rightarrow \Phi_{s_1}$ is realizable. We add $\Psi = \Psi'$ to guarantees of Φ_1 and check if $\Phi_1^{ref} = \Phi_{e_1} \rightarrow (\Phi_{s_1} \wedge \Psi)$ is realizable. If Φ_1^{ref} is not realizable, another assumption refinement for Φ_2 must be computed, and the process is repeated for the new refinement. Note that if adding Ψ to the guarantees of Φ_1 does not make it realizable, there is no Ψ'' such that $\Psi'' \rightarrow \Psi$, and adding Ψ'' keeps Φ_1 realizable. Therefore, a new refinement must be computed.

An advantage of this approach is that the assumption refinement Ψ' for Φ_2 is computed independently using the weakest assumptions that rule out the counter-strategies. Thus, Ψ' can be used even if C_1 is replaced by another component C_1' with different specification, as long as C_1' can still guarantee Ψ'. However, not having enough information about the strategy

Algorithm 3.9: CompositionalRefinement1

Input: $\Phi_1 = \Phi_{e_1} \rightarrow \Phi_{s_1}$: a realizable specification, $\Phi_2 = \Phi_{e_2} \rightarrow \Phi_{s_2}$: an unrealizable specification, α: search depth, U: subset of variables

Output: Ψ such that $\Phi_{e_1} \rightarrow (\Phi_{s_2} \wedge \Psi)$ and $(\Phi_{e_2} \wedge \Psi) \rightarrow \Phi_{s_2}$ are realizable

```
1 while true do
2     Let Ψ := NextAssumptionRefinement(Φ2, α, U);
3     if Ψ = false then
4         break;
5     if Φe1 → (Φs1 ∧ Ψ) is realizable then
6         return Ψ;
7 return No refinement found;
```

computed for C_1, can result in producing counter-strategies that would have not existed if the strategy chosen for C_1 was taken into account. Roughly speaking, the counter-strategy is "irrelevent" with respect to the strategy of C_1. Considering the costly process of generating candidate assumptions for refining the specification, having more knowledge of the strategy of C_1, and hence producing more relevant counter-strategies, might be more desirable, as it decreases the search effort, with the expense of computing an interface specification that is more dependant on the other component's implementation.

Algorithm 3.9 summarizes the first approach. The procedure **NextAssumptionRefinement** is a slightly modified version of Algorithm 3.8 that returns the next possible assumption refinement in the search tree, and `false` if there is no more assumption refinement within the specified depth.

Approach 2 ("Partial knowledge of the strategy of C_1"): For a given counter-strategy, there may exist many different candidate assumptions that can be used to refine the specification. Checking the satisfiability and realizability of the resulting refined specification is an expensive process, so it is more desirable to remove the candidates that are not promising. For example, a counter-strategy might represent a problem that cannot happen due to the strategy chosen by the other component. Roughly speaking, the more one component knows about the other one's implementation, the less number of scenarios it needs to consider and react to.

The second approach shares information about the strategy synthesized for C_1 with C_2 as follows. It computes a set $\mathcal{P}$ of candidate LTL formulas that can be used to refine guarantees of Φ_1. Then at each iteration, a formula $\Psi \in \mathcal{P}$ is chosen, and it is checked if the

Algorithm 3.10: CompositionalRefinement2

Input: $\Phi_1 = \Phi_{e_1} \rightarrow \Phi_{s_1}$: a realizable specification, $\Phi_2 = \Phi_{e_2} \rightarrow \Phi_{s_2}$: an unrealizable specification, α: search depth, U: subset of variables

Output: Refinement Ψ such that $\Phi_{e_1} \rightarrow (\Phi_{s_2} \wedge \Psi)$ and $(\Phi_{e_2} \wedge \Psi) \rightarrow \Phi_{s_2}$ are realizable, or `false` if no such refinement was found

```
1  if α = 0 then
2      return false;
3  Let P = FindGuarantees(Φ2, U);
4  while P is not empty do
5      Remove formula Ψ from P;
6      Let M_CS be the counter-strategy for Φ2;
7      if M_CS ⊨ ¬Ψ then
8          if Φ_e2 ∧ Ψ is satisfiable then
9              if ((Φ_e2 ∧ Ψ) → Φ_s2) is realizable then
10                 return Ψ;
11             else
12                 Let Φ1^new := Φ_e1 → (Φ_s1 ∧ Ψ);
13                 Let Φ2^new := (Φ_e2 ∧ Ψ) → Φ_s2;
14                 Let Ψ' = CompositionalRefinement2(Φ1^new, Φ2^new, α − 1, U);
15                 if Ψ' ≠ false then
16                     return Ψ ∧ Ψ';
17 return false
```

counter-strategy for Φ_2 satisfies $\neg\Psi$ (similar to assumption mining in [LDS11]). If it does and Ψ is consistent with Φ_2, it is checked if Ψ is an assumption refinement for Φ_2, in which case Ψ can be used to refine the guarantees (assumptions) of Φ_1 (Φ_2 , respectively), and Ψ is returned as a suggested refinement. Otherwise, the local specifications are refined by Ψ and the process is repeated with the new specifications. In this approach, some information about C_1's behavior is shared as LTL formulas extracted from the C_1's strategy. Only those formulas that completely rule out the counter-strategy are kept, hence reducing the number of candidate refinements, and keeping the more promising ones, while sharing as much information as needed from one component to the other one. Algorithm 3.10 shows the second approach for computing refinements for input specifications Φ_1 and Φ_2, given the user specified subset of variables U and search depth α.

Approach 3 ("Full knowledge of the strategy of C_1") It might be preferred to refine the specification by adding formulas that are already satisfied by the current imple-

mentation of the realizable component in order not to change the underlying implementation. For example, assume a strategy $\mathcal{M}_S$ is already computed and implemented for Φ_1, and the designer prefers to find a refinement Ψ that is satisfied by $\mathcal{M}_S$. To this end, we can synthesize a set of formulas that hold over all runs of $\mathcal{M}_S$. Any of these formulas can be added as a guarantee to Φ_1, and $\mathcal{M}_S$ will still satisfy the new specification. Yet in some cases, the existing strategy for C_1 must be changed, otherwise C_2 will not be able to fulfill its requirements. In this setting, the guarantees of C_1 can be refined to find a different winning strategy for it.

The third approach is based on this idea. It shares the full knowledge of strategy computed for C_1 with C_2 by encoding the strategy as an LTL formula and providing it as an assumption for Φ_2. Knowing exactly how C_1 plays might make it much easier for C_2 to synthesize a strategy for itself, if one exists. Furthermore, a counter-strategy produced in this case indicates that it is impossible for C_2 to fulfill its goals if C_1 sticks to its current strategy. Therefore, both specifications are refined and a new strategy is computed for the realizable component.

Algorithm 3.11 summarizes the third approach. Once a strategy is computed for the realizable specification, its corresponding LTS $\mathcal{T} = (Q, \{q_0\}, \delta, \mathcal{L})$ is obtained, and encoded as a conjunction of transition formulas as follows. We define a set $\mathcal{Z} = \{z_0, z_1, \cdots, z_{\lceil log|Q| \rceil}\}$ of new propositions that encode the states Q of $\mathcal{T}$. Intuitively, these propositions represent the memory of the strategy in the generated transition formulas, and are considered as environment variables in the refined specification Φ_2'. For ease of notation, let $|\mathcal{Z}|_i$ indicate the truth assignment to the propositions in $\mathcal{Z}$ which represents the state $q_i \in Q$. We encode $\mathcal{T}$ with the conjunctive formula

$$\Psi = (|\mathcal{Z}|_0 \wedge \mathcal{L}(q_0) \wedge \bigwedge_{q_i \in Q} \square((|\mathcal{Z}|_i \wedge \mathcal{L}(q_i)) \rightarrow \bigcirc(\bigvee_{q_j \in \text{Next}(q_i)} |\mathcal{Z}|_j \wedge \mathcal{L}(q_j))$$

where $\text{Next}(q_i)$ is the set of states in $\mathcal{T}$ with a transition from q_i to them. We refer to Ψ as *full encoding* of $\mathcal{T}$. Intuitively, Ψ states that always when the strategy is in state $q_i \in Q$ with truth assignment to the variables given as $\mathcal{L}(q_i)$, then at the next step it will be in one of the adjacent states $q_j \in \text{Next}(q_i)$ with truth assignment $\mathcal{L}(q_j)$ to the variables, and initially it is in state q_0. The procedure **Encode-LTS** in Algorithm 3.11 takes an LTS and returns a conjunctive LTL formula representing it.

Algorithm 3.11: CompositionalRefinement3

Input: $\Phi_1 = \Phi_{e_1} \rightarrow \Phi_{s_1}$: a realizable specification, $\Phi_2 = \Phi_{e_2} \rightarrow \Phi_{s_2}$: an unrealizable specification, α: search depth, U: subset of variables

Output: Ψ such that $\Phi_{e_1} \rightarrow (\Phi_{s_2} \wedge \Psi)$ and $(\Phi_{e_2} \wedge \Psi) \rightarrow \Phi_{s_2}$ are realizable

```
1  if α < 0 then
2      return false;
3  Let S be the strategy for Φ1;
4  Ψ := Encode-LTS(S);
5  Φ2' := (Ψ ∧ Φe2) → Φs2;
6  if Φ2' is realizable then
7      return Ψ;
8  else
9      Let CS' be a counter-strategy for Φ2';
10     P := findCandidateAssumptions(CS', U);
11     foreach Ψ ∈ P do
12         Let Φ2'' be (Ψ ∧ Φe2) → Φs2;
13         Let Φ1'' be Φe1 → (Φs1 ∧ Ψ);
14         if Φ1'' is realizable and Φ2'' is satisfiable then
15             if Φ2'' is realizable then
16                 return Ψ;
17             else
18                 Ψ' := CompositionalRefinement3(Φ1'', Φ2'', α − 1, U);
19                 if Ψ' ≠ false then
20                     return Ψ' ∧ Ψ;
21 return False;
```

Unrealizable specification Φ_2 is then refined by adding the encoding of the strategy as assumptions to it. If the refined specification Φ_2' is realizable, there exists a strategy for C_2, assuming the strategy chosen for C_1, and the encoding is returned as a possible refinement. Otherwise, the produced counter-strategy $\mathcal{CS}'$ shows how the strategy for C_1 can prevent C_2 from realizing its specification. Hence, the specification of both components need to be refined. Procedure **findCandidateAssumptions** uses algorithm 3.8 to compute a set $\mathcal{P}$ of candidate assumptions that can rule out $\mathcal{CS}'$, and at each iteration, one candidate is chosen and tested by both specifications for satisfiability and realizability. If any of these candidate formulas can make both specifications realizable, it is returned as a refinement. Otherwise, the process is repeated with only those candidates that are consistent with Φ_2, and keep Φ_1 realizable. As a result, the set of candidate formulas is pruned, and the process is repeated

with the more promising formulas. If no refinement is found within the specified search depth, `false` is returned.

Remark 3.3. *Introducing new propositions representing the memory of the strategy $\mathcal{S}_1$ computed for Φ_1 leads to assumptions that provide C_2 with full knowledge of how C_1 reacts to its environment. Therefore, if the new specification refined by these assumptions is not realizable, the counter-strategy would be an example of how $\mathcal{S}_1$ might prevent Φ_2 from being realizable, giving the designer the certainty that a different strategy must be computed for C_1, or in other words, both specifications must be refined. However, if introducing new propositions is undesirable, an* abstract encoding *of the strategy (without memory variables) can be obtained by returning conjunction of* all *transition formulas $\square(\psi \rightarrow \bigcirc\psi')$ computed over the strategy. The user can specify the set of variables in which she is interested. This encoding represents an abstraction of the strategy that might be* non-deterministic*, i.e., for the given truth assignment to the environment variables, there might be more than one truth assignment to outputs of C_1 that are consistent with the encoding. Such relaxed encoding can be viewed as sharing partial information about the strategy of C_1 with C_2. As an example, consider the LTS $\mathcal{T}$ in Figure 3.4 which can be encoded as*

$$\begin{aligned}\Psi_{\mathcal{T}} = (q_0 \wedge \neg a \wedge \neg b \wedge \neg c) \wedge \square((q_0 \wedge \neg a \wedge \neg b \wedge \neg c) \rightarrow \bigcirc(q_1 \wedge \neg a \wedge b \wedge \neg c)) \\ \wedge \cdots \wedge \square((q_3 \wedge \neg a \wedge b \wedge \neg c) \rightarrow \bigcirc(q_0 \wedge \neg a \wedge \neg b \wedge \neg c)).\end{aligned}$$

An abstract encoding without introducing new variables and considering only a and b results in formula

$$\begin{aligned}\Psi_{\mathcal{T}}^{a} = \square((\neg a \wedge \neg b) \rightarrow \bigcirc(\neg a \wedge b)) \wedge \square((\neg a \wedge b) \rightarrow \bigcirc((\neg a \wedge \neg b) \vee (a \wedge \neg b)) \\ \wedge \square((a \wedge \neg b) \rightarrow \bigcirc(\neg a \wedge b)).\end{aligned}$$

3.5 Case Study

We now demonstrate the techniques on a robot motion planning case study. We use RATSY [BCG$^+$10] for computing counter-strategies, JTLV [PSZ10] for synthesizing strategies, and Cadence SMV model checker [McM] for model checking. The experiments are performed on a Intel core i7 3.40 GHz machine with 16GB memory.

Consider the robot motion planning example over the discrete workspace shown in Figure

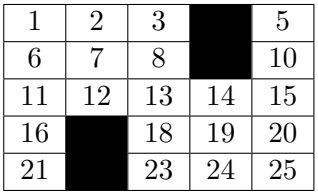


Figure 3.7: Grid-world for the case study

5.1. Assume there are two robots R_1 and R_2 initially in cells 1 and 25, respectively. Robots can move to one of their neighbor cells at each step. There are two rooms in bottom-left and the upper-right corners of the workspace protected by two doors D_1 (cell 10) and D_2 (cell 16). The robots can enter or exit a room through its door and only if it is open. The objective of R_1 (R_2) is to infinitely often visit the cell 5 (21, respectively). The global specification requires each robot to infinitely often visit their goal cells, while avoiding collision with each other, walls and the closed doors, i.e., the robots cannot occupy the same location simultanously, or switch locations in two following time-steps, they cannot move to cells $\{4, 9, 17, 22\}$ (walls), and they cannot move to cells 10 or 16 if the corresponding door is closed. The doors are controlled by the environment and we assume that each door is always eventually open.

The global specification is realizable. We decompose the specification as follows. A local specification $\Phi_1 = \Phi_{e_1} \rightarrow \Phi_{s_1}$ for R_1 where Φ_{e_1} is the environment assumption on the doors and Φ_{s_1} is a conjunction of R_1's guarantees which consist of its initial location, its transition rules, avoiding collision with walls and closed doors, and its goal to visit cell 5 infinitely often. A local specification $\Phi_2 = \Phi_{e_2} \rightarrow \Phi_{s_2}$ for R_2 where Φ_{e_2} includes assumptions on the doors, R_1's initial location, goal, and its transition rules, and Φ_{s_2} consists of R_2's initial location, its transition rules, avoiding collision with R_1, walls and closed doors while fulfilling its goal. The specification Φ_1 is realizable, but Φ_2 is not. We use the algorithms outlined in Section 3.4 to find refinements for both components. We slightly modified the algorithms to find all refinements within the specified search depth. We use the variables corresponding to the location of R_1 for computing the abstraction and generating the candidate formulas. Furthermore, since the counter-strategies are large, computing all eventually and always eventually patterns is not feasible (may take years), and hence we only synthesize some of them using algorithms explained in Section 3.2.

Using the first approach along with abstraction, three refinements are found in 173

minutes which are conjunctions of safety and transition formulas. One of the computed refinements is

$$\begin{aligned}\Psi_1 =&\Box(Loc_{R_1} = 7 \rightarrow \bigcirc(Loc_{R_1} \notin \{7, 8, 12\}))\\ &\wedge\Box(Loc_{R_1} = 13 \rightarrow \bigcirc(Loc_{R_1} \notin \{12, 14\}))\\ &\wedge\Box(Loc_{R_1} = 11 \rightarrow \bigcirc(Loc_{R_1} \neq 16))\\ &\wedge\Box(Loc_{R_1} = 2 \rightarrow \bigcirc(Loc_{R_1} \neq 7))\\ &\wedge\Box(Loc_{R_1} \notin \{2, 12\})\end{aligned}$$

Intuitively, Ψ_1 assumes some restrictions on how R_1 behaves, in which case a strategy for R_2 can be computed. Indeed, R_1 has a strategy that can guarantee Ψ_1. Without using abstraction, four refinements are found within search depth 1 in 17 minutes. A suggested refinement is $\Box(Loc_{R_1} \notin \{7, 12, 16\})$, i.e., if R_1 avoids cells $\{7, 12, 16\}$, a strategy for R_2 can be computed. Using abstraction reduces the number of states of the counter-strategy from 576 to 12 states, however, not all the formulas that are satisfied by the counter-strategy, can be computed over its abstraction, as mentioned in Remark 3.1. Note that computing all the refinements within search depth 3 without using abstraction takes almost 5 times more time compared to when abstraction is used.

Using the second approach (with and without abstraction) the refinement

$$\Psi_2 = \Box(Loc_{R_1} = 10 \rightarrow Loc_{R_1} = 5)$$

is found by inferring formulas from the strategy computed for R_1. Using abstraction slightly improves the process. Finally, using the third approach, providing either the full encoding or the abstract encoding of the strategy computed for Φ_1 as assumptions for Φ_2, makes the specification realizable. Therefore, no counter-strategy is produced, as knowing how R_1 behaves enables R_2 to find a strategy for itself.

Table 3.1 shows the experimental results for the case study. The columns specify the approach, whether abstraction is used or not, the total time for the experiment in minutes, number of strategies (counter-strategies) and number of states of the largest strategy (counter-strategy, respectively), the depth of the search, number of refinements found, and number of candidate formulas generated during the search. As it can be seen from the table, knowing more about the strategy chosen for the realizable specification can significantly

Table 3.1: Evaluation of approaches on a robot motion planning case study

Appr.	Abstr.	time (min)	$\#_{\mathcal{S}}$	max $\lvert Q\rvert_{\mathcal{S}}$	$\#_{\mathcal{CS}}$	max $\lvert Q\rvert_{\mathcal{CS}}$	α	$\#_{ref.}$	$\#_{cand.}$
1	yes	173.05	-	-	17	12	3	3	104
1	no	17.18	-	-	1	576	1	4	22
1	no	869.84	-	-	270	644	3	589	7911
2	yes	69.21	1	8	18	576	1	2	19
2	no	73.78	1	22	19	576	1	2	24
3	yes	0.01	1	8	0	0	1	1	0
3	no	0.02	1	22	0	0	1	1	0

reduce the time needed to find suitable refinement (from hours for the first approach to seconds for the third approach). However, the improvement in time comes with the cost of introducing more coupling between the components, i.e., the strategy computed for C_2 can become too dependent on the strategy chosen for C_1.

4

Compositional Synthesis with Parametric Reactive Controllers

Although automatic synthesis of realistic systems with large state spaces currently appears unattainable, in practice, complex systems are often not constructed from scratch (an implicit assumption in many of the related works,) but from a set of existing building blocks. For example in robot motion planning, a robot usually has a number of predefined motion primitives that can be selected and composed to enforce a high-level objective [FDF05]. Intuitively, a compositional approach that solves smaller and more manageable subproblems, and hierarchically composes the solutions to implement more complicated behaviors seems to be a more plausible way to synthesize complex systems.

To this end, we propose a compositional and hierarchical framework for synthesis from a library of *parametric* and *reactive* controllers. Parameters allow us to take advantage of the symmetry in many synthesis problems, e.g., in motion planning for autonomous robots and vehicles. Reactivity of the controllers takes into account that the environment may be dynamic and potentially adversarial. We first show how these controllers can be synthesized from parametric objectives specified by the user to form a library of parametric and reactive controllers. We then give a synthesis algorithm that selects and instantiates controllers from the library in order to satisfy a given safety and reachability objective.

Consider an autonomous vehicle V_1 that, starting from an initial location s_0, needs to navigate safely through streets and intersections to reach a final destination d, as shown in Figure 4.1. Safe navigation means that the vehicle must follow the traffic rules (e.g., moving in specific directions of streets), and besides avoid collision with other vehicles. In this example, V_1 can cross both intersections I_1 and I_2 on its way toward the location d.

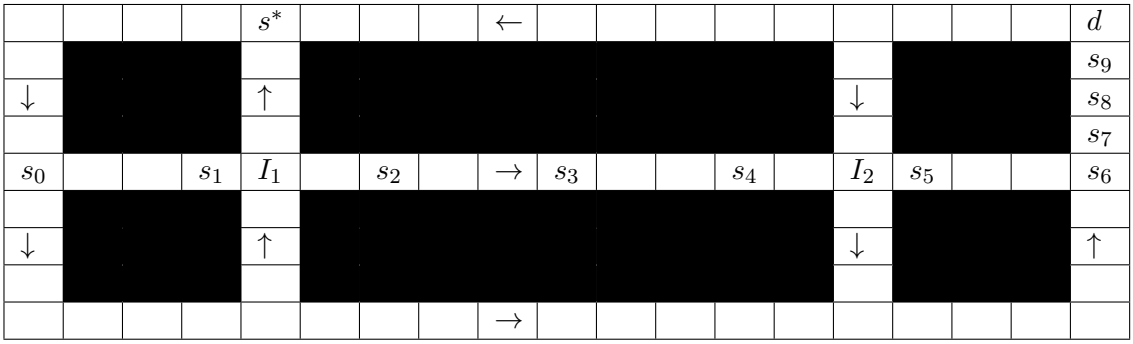


Figure 4.1: One-way streets connected by intersections.

One can observe that although intersections I_1 and I_2 are located in different positions, V_1 can safely cross them in a similar way. In other words, V_1 can employ a controller to cross the intersection I_1 and employ the same controller to cross I_2. To take advantage of such *symmetry* in synthesis problems, we introduce *parametric* controllers. Let (x, y) be the location of V_1 at any time step. Assume a, b are two parameters. We would like to synthesize a controller that starting from a parametric location $(x, y) = (a, b)$, guarantees to eventually move two steps forward horizontally, i.e., eventually $(x, y) = (a + 2, b)$, while avoiding collision with other vehicles. To this end, the parametric controller must also be reactive, i.e., it must react to other vehicles' movements to avoid collision. Once such a parametric reactive controller is obtained, it can be instantiated by assigning values to parameters. For example, the same parametric controller can be instantiated based on the current location of the vehicle and be used to advance the vehicle in different locations. Note that in many application domains, systems may have task-specific controllers that are designed and verified a priori, e.g., an autonomous vehicle can have specialized controllers for different scenarios such as crossing intersections, making U-turns, switching lanes, etc. Such controllers can be defined parametrically and instantiated and composed to perform more complicated tasks.

The proposed framework has two layers, parametric controller synthesis (bottom layer) and synthesis from a library of parametric controllers (top layer). In the bottom layer, a set of parametric controllers are synthesized from parametric objectives specified by the user. Here, unlike other related works [LV13, SRK$^+$14, FDF05], we do not assume that the controllers are a priori given, and we let the user specify them and synthesis is done automatically. This flexibility facilitates the design process, allowing the user to utilize her

insight into the system being designed to construct different libraries. Furthermore, the user may not know the range of the parameter values that guarantees correct behavior of the controller. We allow the user to provide a parametric specification and the set of acceptable parameter values are discovered automatically. On the other hand, the high-level composer does not necessarily need to know *how* controllers enforce their objectives. Thus a *controller interface* that hides the controller's specific implementation while providing information on possible outcomes of the controller is synthesized for each parametric controller. A library of parametric controllers can be reused to realize more complex behaviors. In the top layer of the framework, given a library of parametric controllers and a high-level objective for the system, a *control strategy* that selects and instantiates parametric controllers from the library such that their composition enforces the objective is synthesized.

Note that adding parameters increases the size of the state space and can add to the complexity of the problem. Therefore, how parameters are handled is crucial. We provide *symbolic* algorithms that efficiently explore the parametric space. Besides, we show that the upper bound on the number of symbolic steps, i.e., pre-image or post-image computations, performed by the symbolic algorithm is independent from the parameters. Nevertheless, this does not mean that adding parameters has no cost as it increases the complexity of the symbolic steps. The main advantages of the introduced framework are twofold: *i*) *Reusability of controllers* (parametric controllers are computed once and can be reused in different compositions to achieve higher level objectives), *ii*) *Separation of concerns* (design of controllers is separated from their composition which can also lead to strategies that are defined hierarchically and are easier to understand).

The concept of *motion primitives* is popular and widely used in robotics and control literature, since they can be designed by one group, e.g., the robot designer, and then be used by other groups of people such as the end-users to implement higher level objectives. The end-user only needs to have an understanding of what a specific motion primitive does through a provided interface, and the actual implementation is encapsulated and hidden from the end-user. A compositional motion planning framework for multi-robot systems is presented in [SRK$^+$14] where given a library of motion primitives, the motion planning problem is reduced to solving a satisfiability modulo theories problem. A similar approach to ours is considered in [FDF05] for solving motion-planning problems for time-invariant dynamical control systems with symmetries, such as mobile robots and autonomous vehicles,

where motion plans are described as concatenation of a number of motion-primitives chosen from a finite library. The main difference of our work with [FDF05, SRK$^+$14] is that our motion primitives are *reactive*, i.e., the controllers also takes the ongoing interaction between system and environment into account. To the best of our knowledge, we are the first to study the problem of synthesizing controllers from a library of parametric and reactive controllers.

The problem of LTL synthesis from a library of reusable components is considered in [LV13]. Sequential composition of controllers considered in this chapter is similar to control-flow composition in [LV13] and is inspired by software systems. In the software context, when a function is called, the function gains the control over the machine and the computation proceeds according to the function until it calls another function or returns. Similarly, the controllers in our framework gain and relinquish control over computations of the system. The controllers have a designated set of final states. Intuitively, a reactive controller receives the control by entering an initial state and returns the control when reaching a final state. The goal of the composer is to decide which controller will gain control when the control is returned from the controller currently in charge. Although by enumerating the parameter values and instantiating parametric controllers to obtain a library of non-parametric controllers our problem can be reduced to the one considered in [LV13], such naive enumeration may lead to an exponentially larger number of controllers in the library, making the method infeasible in practice. Our algorithms *symbolically* explore the parametric space, thus avoiding the excessive explicit enumeration.

4.1 Controllers, Controller Interfaces, and Sequential Composition

Controllers are building blocks of the proposed framework in this chapter. Given a high-level objective, the composer selects and instantiates the appropriate controllers using the information provided through their interfaces. Let $\mathbb{Z}$ be the set of integers. For $a, b \in \mathbb{Z}$, let $[a..b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$. Let $\mathcal{P} = \{p_1, \cdots, p_k\}$ be a set of parameters where each parameter $p \in \mathcal{P}$ is defined over a finite domain Σ_p. We define $\Sigma_{\mathcal{P}} = \Sigma_{p_1} \times \cdots \times \Sigma_{p_k}$ to be the collective domain of the parameters.

For a predicate ϕ, let $\mathcal{VP}(\phi)$ be the set of variables and parameters that appear in the

predicate's formula. We say ϕ is a *parametric* predicate, if $\mathcal{VP}(\phi) \cap \mathcal{P} \neq \emptyset$, i.e., there is at least one parameter in the predicate's formula. Otherwise we say ϕ is non-parametric. Given a parametric predicate ϕ over $\mathcal{V} \cup \mathcal{P}$ and a valuation $p \in \Sigma_{\mathcal{P}}$ over parameters, restriction of ϕ by p is a non-parametric predicate $\phi_{\downarrow p}$ obtained by replacing each parameter with its corresponding value. Given a parametric set $\Pi = \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{P}}$ and a parameter value $p \in \Sigma_{\mathcal{P}}$, projection of Π by p, denoted by $\Pi_{\downarrow p}$, is the set $\{s \in \Sigma_{\mathcal{V}} \mid (s, p) \in \Pi\}$.

Without loss of generality and to simplify the specification language, we assume that all variables and parameters are defined over bounded integer domains in the rest of this chapter. Boolean variables are special case where the domain is $\{0, 1\}$. Note that since the domains of variables and parameters are finite, they can be encoded using Boolean variables. Ordered binary decision diagrams (OBDDs) can be used for obtaining concise representations of sets and relations over finite domains [CGP99]. Before formally stating the problems that are considered in this chapter, we provide some definitions.

Controller. We refer to memory-less strategies for player-2 with a designated set of final states as *finite-horizon reactive controllers* (or *controllers* for short). In our setting, controllers receive control of the system for a finite number of steps and interact with environment until reaching a desirable target state while avoiding some specified error states. Formally, a controller $\mathcal{C}$ is a pair $(\mathcal{S}, \mathcal{F})$ where $\mathcal{S} : \Sigma_{\mathcal{V}} \to \Lambda$ is a memory-less strategy and $\mathcal{F} \subseteq \Sigma_{\mathcal{V}}$ is a designated set of final states. At any time-step, if current state $s \in \Sigma_{\mathcal{V}}$ is a final state, i.e., $s \in \mathcal{F}$, the controller has reached the end of its computation. Note that we only consider controllers with reachability and safety objectives for which memory-less strategies suffice. A parametric reactive controller is a controller whose strategy and set of final states are parametric. Given a parameter valuation $p \in \Sigma_p$ and a parametric controller $\mathcal{C} = (\mathcal{S}, \mathcal{F})$, instantiation of $\mathcal{C}$ with $p \in \mathcal{P}$ is the controller $\mathcal{C}_{\downarrow p} = (\mathcal{S}_{\downarrow p}, \mathcal{F}_{\downarrow p})$ obtained by instantiating the strategy and projecting the set of final states by p.

Controller interface. A controller interface abstracts a controller by providing high-level information about its behavior while hiding the actual implementation and executions of the controller. Formally, a controller interface $\mathcal{I}_{\mathcal{C}} = (\phi_{init_{\mathcal{C}}}, \phi_{inv_{\mathcal{C}}}, \phi_{f_{\mathcal{C}}})$ for a controller $\mathcal{C}$ is a tuple where $\phi_{init_{\mathcal{C}}}$ is a set of initial valuations over variables (and parameters), $\phi_{inv_{\mathcal{C}}}$ is an invariant that holds over all possible runs of $\mathcal{C}$ while it has the control, $\phi_{f_{\mathcal{C}}}$ is a possible set of valuations over variables (and parameters) once $\mathcal{C}$ reaches a final state. A controller $\mathcal{C} = (\mathcal{S}, \mathcal{F})$ over a game structure $\mathcal{G}$ *realizes* a controller interface $\mathcal{I}_{\mathcal{C}}$ if $\mathcal{S}$ is a winning

strategy for the game $(\mathcal{G}, \phi_{init_\mathcal{C}}, \Psi)$ where $\Psi = \phi_{inv_\mathcal{C}}\ \mathcal{U}\ (\phi_{inv_\mathcal{C}} \wedge \phi_{f_\mathcal{C}})$ and $\mathcal{F} \subseteq [\![\phi_{f_\mathcal{C}}]\!]$. That is, starting from any initial state $v \models \phi_{init_\mathcal{C}}$, controller $\mathcal{C}$ guarantees that eventually a final state $v_f \models \phi_{f_\mathcal{C}}$ is visited and besides all the visited states along any possible outcome satisfy $\phi_{inv_\mathcal{C}}$, i.e., only safe states are visited. Instantiation of a controller interafce $\mathcal{I}_\mathcal{C}$ by $p \in \Sigma_\mathcal{P}$ is the non-parametric controller interface $\mathcal{I}_{\mathcal{C}_{\downarrow p}} = (\phi_{init_{\mathcal{C}_{\downarrow p}}}, \phi_{inv_{\mathcal{C}_{\downarrow p}}}, \phi_{f_{\mathcal{C}_{\downarrow p}}})$. A parameter valuation $p \in \Sigma_\mathcal{P}$ is *admissible* for controller $\mathcal{C}$ with interface $\mathcal{I}_\mathcal{C}$ over a game structure $\mathcal{G}$ if and only if the instantiation of $\mathcal{C}$ by p, $\mathcal{C}_{\downarrow p}$, realizes the non-parametric interface $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$. Intuitively, a parametric controller can be instantiated by any admissible parameter value, and enforce its safety and reachability objectives, provided that its execution starts from a valid initial state. A set $\Sigma^a_\mathcal{P} \subseteq \Sigma_\mathcal{P}$ of admissible parameter values is maximal, if for any parameter valuation $p \in \Sigma_\mathcal{P} \backslash \Sigma^a_\mathcal{P}$, $\mathcal{C}_{\downarrow p}$ does *not* realize $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$. A controller interface $\mathcal{I}_1 = (\phi_{init_1}, \phi_{inv_1}, \phi_{f_1})$ *respects* a controller interface $\mathcal{I}_2 = (\phi_{init_2}, \phi_{inv_2}, \phi_{f_2})$ if $\phi_{init_1} \rightarrow \phi_{init_2}$, $\phi_{inv_1} \rightarrow \phi_{inv_2}$, and $\phi_{f_1} \rightarrow \phi_{f_2}$. It is easy to see that any controller that realizes $\mathcal{I}_1$ also realizes the restricted interface $\mathcal{I}'_2 = (\phi_{init_1}, \phi_{inv_2}, \phi_{f_2})$. Note that $\mathcal{I}'_2$ is obtained from the interface $\mathcal{I}_2$ by restricting its initial states to $[\![\phi_{init_1}]\!] \subseteq [\![\phi_{init_2}]\!]$. In our setting, the designer can specify a parametric interface for the controllers without knowing for what parameter valuations the controller can enforce its safety and reachability objectives. A parametric controller, a maximal set of admissible parameter values, and an interface that respects the user-specified interface are then synthesized automatically.

Composing Controllers Let $\mathcal{G} = (\mathcal{V}, \Lambda, \tau)$ be a game structure, and $\Gamma_\mathcal{C} = \{\mathcal{C}_1, \cdots \mathcal{C}_n\}$ be a set of parametric controllers. For a given set of initial states ϕ_{init} and objective Φ, the goal of the composer is to iteratively select a parametric controller and instantiate it with a parameter valuation, delegate the control to the instantiated controller until it enters a final state and relinquishes the control, upon which the composer selects the next controller and the next parameter valuation, and the process is repeated such that the objective Φ is enforced starting from any initial state $v_{init} \models \phi_{init}$. A *control strategy* $\mathcal{S}^\mathcal{C} : \Sigma_\mathcal{V} \rightarrow \Sigma_\mathcal{P} \times \Gamma_\mathcal{C}$ is a (partial) function that maps states of the game to a controller and a parameter valuation (note that we do not consider memory for the control strategy since it is not needed for safety and reachability objectives). A control strategy $\mathcal{S}^\mathcal{C}$ induces a finite-memory strategy $\mathcal{S} = (m_0, f_M, f_\Lambda)$ obtained by *sequentially* composing instantiated controllers according to $\mathcal{S}^\mathcal{C}$ as follows. Let $M \subseteq \Sigma_\mathcal{P} \times \Gamma_\mathcal{C} \cup \{\perp\}$ be the memory of the strategy where $m_0 = \perp$ and $\perp$ is a special symbol indicating the initial memory where a controller and a parameter valuation

is yet to be selected. Intuitively, the memory of the strategy keeps track of the controller that currently has the control and the parameter valuation used to instantiate it. The memory-update function $f_M : M \times \Sigma_{\mathcal{V}} \to M$ and the next-action function $f_\Lambda : M \times \Sigma_{\mathcal{V}}^2 \to \Lambda$ are defined as

$$f_M(m, v) = \begin{cases} m & \text{if } m \neq \perp \wedge v \notin \mathcal{F}_{\mathcal{C}_m} \\ \mathcal{S}^{\mathcal{C}}(v) & \text{otherwise} \end{cases}$$

$$f_\Lambda(m, v) = \begin{cases} \mathcal{S}_{\mathcal{C}_m}(v) & \text{if } m \neq \perp \wedge v \notin \mathcal{F}_{\mathcal{C}_m} \\ \mathcal{S}_{\mathcal{C}_{next}}(v) & \text{otherwise} \end{cases}$$

where $\mathcal{C}_m = (\mathcal{S}_{\mathcal{C}_m}, \mathcal{F}_{\mathcal{C}_m}) = \mathcal{C}_{i\downarrow p}$ is the instantiated controller for the memory $m = (p, \mathcal{C}_i)$, and $\mathcal{C}_{next} = (\mathcal{S}_{\mathcal{C}_{next}}, \mathcal{F}_{\mathcal{C}_{next}}) = \mathcal{C}^{next}_{\downarrow p^{next}}$ with $\mathcal{S}^{\mathcal{C}}(v) = (p^{next}, \mathcal{C}^{next})$ is the next controller chosen by the control strategy. Intuitively, when a final state of the currently active controller is reached or initially when no controller is selected, the next controller and the next parameter valuation are chosen according to the control strategy and the memory is updated to reflect this selection. The selected and instantiated controller then becomes active and guides the actions of the system while the memory stays unchanged, until the active controller enters a final state, upon which the control strategy decides the next action and the process is repeated. Note that in practice, the induced strategy $\mathcal{S}$ from $\mathcal{S}^{\mathcal{C}}$ is not computed explicitly, and the controllers can be dynamically fetched, instantiated and executed according to the control strategy.

4.2 Problem Statement and Overview

We are now ready to formally define the problems considered in this chapter and give an overview of our solution approach. Let $\mathcal{V}$ and $\mathcal{P}$ be sets of variables and parameters defined over finite domains $\Sigma_{\mathcal{V}}$ and $\Sigma_{\mathcal{P}}$, respectively, Λ be a finite set of actions, and $\mathcal{G}$ be a game structure over $\mathcal{V}$ and Λ. We are interested in how a parametric controller can be synthesized from a given parametric controller interface. Formally,

Problem Statement 4.1. *(Synthesis of Parametric Reactive Controllers.) Given a game structure $\mathcal{G}$ and a parametric controller interface $\mathcal{I} = (\phi_{init}, \phi_{inv}, \phi_f)$, synthesize a parametric reactive controller $\mathcal{C}$, its corresponding interface $\mathcal{I}_{\mathcal{C}}$, and a maximal set $\Sigma^a_{\mathcal{P}} \subseteq \Sigma_{\mathcal{P}}$ of admissible*

$(0,0)$	$(1,0)$	$(2,0)$	$(3,0)$	$(4,0)$	$(5,0)$	$(6,0)$	$(7,0)$
$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$	$(5,1)$	$(6,1)$	$(7,1)$

Figure 4.2: Part of a road divided into grids.

parameter valuations such that $\mathcal{I}_{\mathcal{C}}$ respects $\mathcal{I}$, and for any admissible parameter valuation $p \in \Sigma^a_{\mathcal{P}}$ for $\mathcal{C}$, instantiation of $\mathcal{C}$ by p, $\mathcal{C}_{\downarrow p}$, realizes the instantiated controller interface $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$.

A designer can specify a set of parametric controller interfaces. The synthesis algorithm then automatically computes the set of controllers, their corresponding interfaces and admissible parameter values. Once the parametric controllers are computed, they can be reused in different compositions to synthesize control strategies for different objectives.

Once a library of parametric controllers and their corresponding interfaces are obtained, the next natural question is how they can be composed to enforce high-level objectives. Let ϕ_{init} be a non-parametric predicate specifying initial states of the game, Φ be a non-parametric LTL objective over $\mathcal{V}$, $\Gamma_{\mathcal{C}} = \{\mathcal{C}_1, \cdots, \mathcal{C}_n\}$ be a set of parametric controllers, and $\Gamma_{\mathcal{I}_{\mathcal{C}}} = \{\mathcal{I}_{\mathcal{C}_1}, \cdots, \mathcal{I}_{\mathcal{C}_n}\}$ be the set of corresponding controller interfaces. Our goal is to synthesize a control strategy $\mathcal{S}^{\mathcal{C}}$ that instantiates and composes controllers from $\Gamma_{\mathcal{C}}$ using the information provided through interfaces $\Gamma_{\mathcal{I}_{\mathcal{C}}}$ such that its induced strategy enforces the global objective Φ in the game $(\mathcal{G}, \phi_{init}, \Phi)$. Formally,

Problem Statement 4.2. *(Synthesis with Parametric Reactive Controllers.) Given a game structure $\mathcal{G}$, a set of initial states specified by a non-parametric predicate ϕ_{init}, a non-parametric LTL objective Φ, and a set of parametric controllers $\Gamma_{\mathcal{C}}$ and their corresponding interfaces $\Gamma_{\mathcal{I}_{\mathcal{C}}}$, compute a control strategy $\mathcal{S}^{\mathcal{C}}$, if one exists, such that its induced strategy $\mathcal{S}$ is winning in the game $(\mathcal{G}, \phi_{init}, \Phi)$.*

We assume that Φ is given as a safety and/or reachability objective. We illustrate the methods with a simple example.

Example 4.1. *Consider a block of a double-lane road divided into grids each identified by a tuple (x, y) as shown in Figure 4.2. Assume there is a controlled vehicle V_1 initially at $(x_1, y_1) = (0, 1)$ moving from left to right. Moreover, assume there is an uncontrolled vehicle V_2 initially at $(x_2, y_2) = (7, 1)$ moving from right to left while staying on the same lane at all times, i.e., always $y_2 = 1$. Formally, let $\phi_{init} = (x_1 = 0 \wedge y_1 = 1 \wedge x_2 = 7 \wedge y_2 = 1)$ be the*

predicate specifying the initial state of the system. Assume V_1 has two actions: move-forward action, ℓ_1, that moves the vehicle one step ahead by incrementing x_1 while keeping it on the same lane, and lane-switch action, ℓ_2, that moves the vehicle one step forward and changes the lane at the same time. Our goal is to synthesize a controller that guides V_1 from the starting point to the other end of the road without colliding with V_2. This objective can be specified with the formula $\Phi = \phi_1 \, \mathcal{U} \, (\phi_1 \wedge \phi_2)$ where $\phi_1 = (x_1 \neq x_2 \vee y_1 \neq y_2)$ (no collision) and $\phi_2 = (x_1 = 7)$ (reaching the other end.)

Let a and b be two parameters. Assume the designer specifies a parametric controller interface $\mathcal{I} = (\phi_{init}, \phi_{inv}, \phi_f)$ where $\phi_{init} = (x_1 = a) \wedge (y_1 = b)$, $\phi_{inv} = (x_1 \neq x_2) \vee (y_1 \neq y_2)$, and $\phi_f = (x_1 = a + 1)$, i.e., starting from initial parametric state $(x_1, y_1) = (a, b)$, V_1 must move one step forward (to satisfy ϕ_f) while avoiding collision with V_2 (thus satisfying ϕ_{inv}). A parametric controller $\mathcal{C} = (\mathcal{S}, \mathcal{F})$ is then synthesized with a memory-less strategy $\mathcal{S}$ defined as

$$\mathcal{S}(x_1, y_1, x_2, y_2, a, b) = \begin{cases} \ell_2 \quad \textit{if} \quad 0 \leq a \leq 6 \wedge x_1 = a \wedge y_1 = b \\ \qquad\quad \wedge y_1 = y_2 \wedge x_2 = a + 1 \\ \ell_1 \quad \textit{if} \quad 0 \leq a \leq 6 \wedge x_1 = a \wedge y_1 = b \wedge \\ \qquad\quad (x_1 \neq x_2 \vee (y_2 \neq b \wedge y_2 \neq b + 1)) \end{cases}$$

Intuitively, the controller $\mathcal{C}$ switches the current lane of the vehicle V_1 by taking lane-switch action ℓ_2 if the other vehicle V_2 is on the same lane and one cell ahead of V_1, and otherwise keeps moving forward by taking move-forward action ℓ_1. This way the controller $\mathcal{C}$ ensures that V_1 eventually makes progress by incrementing x_1 while avoiding collision with the other vehicle. For the set of final states of $\mathcal{C}$ we have $\mathcal{F} = (0 \leq a \leq 6 \wedge x_1 = a + 1 \wedge ((x_1 \neq x_2) \vee (y_1 \neq y_2)))$, i.e., once the controller reaches a final state, V_1 has moved one step forward and does not occupy the same grid with V_2. Besides, correct behavior of the controller is guaranteed for the parameter values $0 \leq a \leq 6$. A potential controller interface $\mathcal{I}_\mathcal{C}$ for $\mathcal{C}$ is $(\phi'_{init}, \phi_{inv}, \phi_f)$ where $\phi'_{init} = \phi_{init} \wedge 0 \leq a \leq 6$. Note that $\mathcal{I}_\mathcal{C}$ respects $\mathcal{I}$ and $\mathcal{C}$ realizes $\mathcal{I}_\mathcal{C}$.

Once the parametric controllers are synthesized and a library is formed, the next step is to instantiate right parametric controllers and compose them to enforce a given system objective. In the above example, the controller $\mathcal{C}$ can be instantiated and composed sequentially in order to enforce the objective Φ according to the memory-less control strategy $\mathcal{S}^\mathcal{C}(x_1, y_1, x_2, y_2) = ((x_1, y_1), \mathcal{C}) \quad$ if $0 \leq x_1 \leq 6$. Intuitively, while V_1 has not reached the end of the road (i.e.,

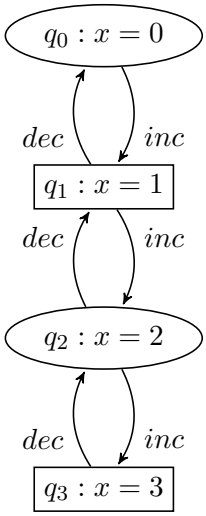


Figure 4.3: A game structure $\mathcal{G}$ defined over a variable $x \in [0..3]$.

$x_1 \neq 7$), the control strategy selects $\mathcal{C}$ and instantiates it with $(a = x_1, b = y_1)$, i.e., V_1's current location. To enforce the objective Φ, the parametric controller $\mathcal{C}$ is instantiated and composed 7 times, where each controller moves the vehicle one step forward without colliding with the other vehicle.

4.3 Synthesizing Parametric Reactive Controllers

In this section we describe our solution for Problem 4.1 stated in Section 4.2. Let $\mathcal{G} = (\mathcal{V}, \Lambda, \tau)$ be a game structure, and $\mathcal{I} = (\phi_{init}, \phi_{inv}, \phi_f)$ be the user-specified controller interface. Our goal is to synthesize a controller $\mathcal{C}$ and its corresponding controller interface $\mathcal{I}_\mathcal{C} = (\phi_{init_\mathcal{C}}, \phi_{inv_\mathcal{C}}, \phi_{f_\mathcal{C}})$ and a set $\Sigma^a_\mathcal{P}$ of admissible parameter values such that for any $p \in \Sigma^a_\mathcal{P}$, $\mathcal{C}_{\downarrow p}$ realizes $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$ and $\mathcal{I}_\mathcal{C}$ respects $\mathcal{I}$.

To this end, we first obtain a *parametric* game structure $\mathcal{G}^\mathcal{P}$ from $\mathcal{G}$. The idea is to treat parameters as special variables that have unknown initial value in a bounded set, but their value stays constant over the transitions of the game structure. Formally, let $\mathcal{P}'$ be a primed copy of parameters, and assume $same(\mathcal{P}, \mathcal{P}')$ is a predicate stating that the value of parameters stay unchanged. The parametric game structure $\mathcal{G}^\mathcal{P}$ is defined as $(\mathcal{V} \cup \mathcal{P}, \Lambda, \tau^\mathcal{P})$ where $\tau^\mathcal{P} = \tau \wedge same(\mathcal{P}, \mathcal{P}')$.

For example, Figure 4.3 shows a game structure where player-1 (player-2) states are depicted by ovals (boxes, respectively.) Each state is labeled by a state name q_i and a valuation over a variable x with domain $\Sigma_x = [0..3]$. At each player-i state for $i = 1, 2$, the

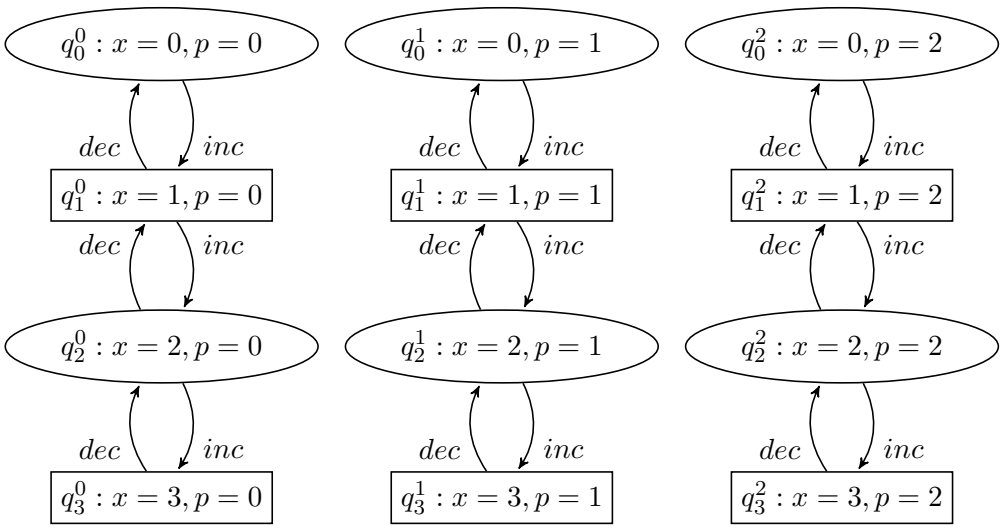


Figure 4.4: A parametric game structure $\mathcal{G}^{\mathcal{P}}$ obtained from $\mathcal{G}$ with parameter $p \in [0..2]$.

player can choose one of the actions *inc* or *dec* (if available,) to increment or decrement x, respectively. Assume p is a parameter with domain $\Sigma_p = [0..2]$. Figure 4.4 shows the parametric game structure obtained from the game structure in Figure 4.3. Each state is labeled with a state name q_i^j and a valuation over x and p. Each state q_i^j in the parametric game structure $\mathcal{G}^{\mathcal{P}}$ correspond to the state q_i in the game structure $\mathcal{G}$. Intuitively, the parametric game structure has parallel copies of the non-parametric game structure for different values of the parameters and moreover, there is no transition between different copies. Note that explicit-state representations of (parametric) game structures are *not* constructed in practice, and they are represented and manipulated symbolically, thus avoiding the explicit enumeration of the parameters.

Algorithm 4.1 shows how a parametric controller is synthesized for a given game structure $\mathcal{G}$ and specified interface $\mathcal{I}$. Once the parametric game structure $\mathcal{G}^{\mathcal{P}}$ is obtained, the game $(\mathcal{G}^{\mathcal{P}}, \phi_{init}, \Phi_{\mathcal{I}})$, where $\Phi_{\mathcal{I}} = \phi_{inv}\ \mathcal{U}\ (\phi_{inv} \wedge \phi_f)$, can be solved by standard realizability and synthesis algorithms and a set of winning states can be computed [MPS95]. Let $\mathcal{W} \subseteq \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{P}}$ be the set of winning states in $\mathcal{G}^{\mathcal{P}}$ with respect to objective $\Phi_{\mathcal{I}}$, and let $\phi_{\mathcal{W}}$ be a predicate specifying $\mathcal{W}$, i.e., $[\![\phi_{\mathcal{W}}]\!] = \mathcal{W}$. We define $\phi_{init_c} = \phi_{init} \wedge \phi_{\mathcal{W}}$ as the intersection of set of parametric initial states specified by the user and set of winning states where player-2 can enforce the objective $\Phi_{\mathcal{I}}$. The set $[\![\phi_{init_c}]\!]$ includes all the parametric initial states from which player-2 can win the game $(\mathcal{G}^{\mathcal{P}}, \phi_{init_c}, \Phi_{\mathcal{I}})$ and hence, it contains all the admissible parameter valuations. The set $\Sigma_{\mathcal{P}}^a$ of admissible parameter values can

Algorithm 4.1: Parametric Controller Synthesis

Input: Game structure $\mathcal{G} = (\mathcal{V}, \Lambda, \tau)$, controller interface $\mathcal{I} = (\phi_{init}, \phi_{inv}, \phi_f)$ and parameters $\mathcal{P}$

Output: Parametric controller $\mathcal{C}$, controller interface $\mathcal{I}_\mathcal{C}$, and admissible parameter values $\Sigma^a_\mathcal{P}$ s.t. $\mathcal{I}_\mathcal{C}$ respects $\mathcal{I}$ and $\forall p \in \Sigma^a_\mathcal{P}$. $\mathcal{C}_{\downarrow p}$ realizes $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$.

1 $\tau^\mathcal{P} := \tau \wedge same(\mathcal{P}, \mathcal{P}')$;
2 $\mathcal{G}^\mathcal{P} := (\mathcal{V} \cup \mathcal{P}, \Lambda, \tau^\mathcal{P})$;
3 $\Phi_\mathcal{I} := \phi_{inv}\ \mathcal{U}\ (\phi_{inv} \wedge \phi_f)$;
4 Let $[\![\phi_\mathcal{W}]\!]$ be the set of winning states in $\mathcal{G}^\mathcal{P}$ with respect to $\Phi_\mathcal{I}$;
5 $\phi_{init_\mathcal{C}} := \phi_{init} \wedge \phi_\mathcal{W}$;
6 $\phi^a_\mathcal{P} := \exists \mathcal{V}.\ \phi_{init_\mathcal{C}}$;
7 $\Sigma^a_\mathcal{P} := [\![\phi^a_\mathcal{P}]\!]$;
8 Let $\mathcal{S}$ be a parametric winning strategy in the game $(\mathcal{G}^\mathcal{P}, \phi_{init_\mathcal{C}}, \Phi_\mathcal{I})$;
9 $\phi_\mathcal{R} := \textbf{Reachable}(\mathcal{G}^\mathcal{P}, \phi_{init_\mathcal{C}}, \mathcal{S})$;
10 $\mathcal{F} := [\![\phi_f \wedge \phi_\mathcal{R}]\!]$;
11 $\mathcal{C} := (\mathcal{S}, \mathcal{F})$;
12 $\phi_{inv_\mathcal{C}} := \phi_\mathcal{R}$;
13 $\phi_{f_\mathcal{C}} := \phi_f \wedge \phi_\mathcal{R}$;
14 $\mathcal{I}_\mathcal{C} = (\phi_{init_\mathcal{C}}, \phi_{inv_\mathcal{C}}, \phi_{f_\mathcal{C}})$;
15 return $(\mathcal{C}, \mathcal{I}_\mathcal{C}, \Sigma^a_\mathcal{P})$;

be computed by existentially quantifying the variables from $\phi_{init_\mathcal{C}}$, i.e., $\Sigma^a_\mathcal{P} = \exists \mathcal{V}.\ \phi_{init_\mathcal{C}}$. Algorithm 4.1 then computes a parametric winning strategy over the game $(\mathcal{G}^\mathcal{P}, \phi_{init_\mathcal{C}}, \Phi_\mathcal{I})$ using a game solver. Let $\phi_\mathcal{R} = \textbf{Reachable}(\mathcal{G}^\mathcal{P}, \phi_{init_\mathcal{C}}, \mathcal{S})$ be a predicate specifying the set $[\![\phi_\mathcal{R}]\!] \subseteq \Sigma_\mathcal{V} \times \Sigma_\mathcal{P}$ of reachable states in the parametric game structure $\mathcal{G}^\mathcal{P}$ starting from any initial state $s \models \phi_{init_\mathcal{C}}$ when player-2 actions are chosen according to the strategy $\mathcal{S}$. We define $\mathcal{I}_\mathcal{C} = (\phi_{init_\mathcal{C}}, \phi_{inv_\mathcal{C}}, \phi_{f_\mathcal{C}})$ as the controller interface for $\mathcal{C}$ where $\phi_{inv_\mathcal{C}} = \phi_\mathcal{R}$ and $\phi_{f_\mathcal{C}} = \phi_f \wedge \phi_\mathcal{R}$. Intuitively, $\phi_{inv_\mathcal{C}}$ specifies the set of states that may be visited during the game when player-2 behaves according to the controller $\mathcal{C}$, and serves as the invariant of the computed controller interface. Similarly, $\phi_{f_\mathcal{C}}$ specifies a set of reachable final states when $\mathcal{C}$ is active. The following theorem states that Algorithm 4.1 can correctly synthesize a parametric controller, if one exists, and that it computes the controller with effort $O(|\Sigma_\mathcal{V}|)$, where effort is measured in symbolic steps, i.e., in the number of pre-image or post-image computation [BGS06].

Theorem 4.1. *Algorithm 4.1 is sound and complete. It performs $O(|\Sigma_\mathcal{V}|)$ symbolic steps in the worst case.*

Proof. We first show that Algorithm 4.1 is sound. We need to show that $\mathcal{I}_\mathcal{C}$ respects $\mathcal{I}$,

and for any parameter valuation $p \in \Sigma^a_{\mathcal{P}}$ for $\mathcal{C}$, $\mathcal{C}_{\downarrow p}$ realizes $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$. Note that the parametric transition relation $\tau^{\mathcal{P}}$ has two components: τ, the transition relation of the game structure $\mathcal{G}$, that is independent from parameters, and $same(\mathcal{P}, \mathcal{P}')$ that is independent from variables $\mathcal{V}$. Therefore, any run $\pi^{\mathcal{P}}$ in the parametric game structure $\mathcal{G}^{\mathcal{P}}$ is of the form $\pi^{\mathcal{P}} = (v_0, p)(v_1, p)(v_2, p) \cdots$ where $v_i \in \Sigma_{\mathcal{V}}$ for $i \geq 0$ and $p \in \Sigma_{\mathcal{P}}$. Also note that $\pi^{\mathcal{P}}$ corresponds to a run $\pi = v_0 v_1 v_2 \cdots$ in the game structure $\mathcal{G}$ where the parameters are simply stripped away. Line 4 of algorithm 4.1 computes the set $[\![\phi_{\mathcal{W}}]\!]$ of *all* winning states in the parametric game structure $\mathcal{G}^{\mathcal{P}}$ with respect to the objective $\phi_{\mathcal{I}}$. Thus, $\phi_{init_{\mathcal{C}}}$ given as intersection of the specified parametric initial states ϕ_{init} and the set of winning states $\phi_{\mathcal{W}}$ from which the objective $\phi_{\mathcal{I}}$ can be enforced, represents the largest subset of $[\![\phi_{init}]\!]$ that is winning. We have $[\![\phi_{init_{\mathcal{C}}}]\!] = \{(v, p) \in \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{P}} \mid (v, p) \models \phi_{init} \wedge \phi_{\mathcal{W}}\}$. In lines 6 and 7 in Algorithm 4.1 the set $\Sigma^a_{\mathcal{P}}$ of admissible parameter valuations is computed as $\Sigma^a_{\mathcal{P}} = \{p \in \Sigma_{\mathcal{P}} \mid \exists v \in \Sigma_{\mathcal{V}}.(v, p) \models \phi_{init_{\mathcal{C}}}\}$. Let $p \in \Sigma^a_{\mathcal{P}}$ be a parameter valuation. Let $v_0 \in \Sigma_{\mathcal{V}}$ be any valuation over variables $\mathcal{V}$ such that $(v_0, p) \models \phi_{init_{\mathcal{C}}}$. By definition of $\Sigma^a_{\mathcal{P}}$ such v_0 exists. Let $\mathcal{C} = (\mathcal{S}, \mathcal{F})$ be the controller computed by Algorithm 4.1. We show that for any $p \in \Sigma^a_{\mathcal{P}}$, the controller $\mathcal{C}_{\downarrow p}$ realizes $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$. Let $\pi^p = (v_0, p)(v_1, p) \cdots (v_f, p) \cdots$ be any run in the game structure $\mathcal{G}^{\mathcal{P}}$ where $(v_0, p) \models \phi_{init_{\mathcal{C}}}$ and actions of player-2 are chosen according to the parametric strategy $\mathcal{S}$. Since $\mathcal{S}$ is a winning strategy in the game $(\mathcal{G}^{\mathcal{P}}, \phi_{init_{\mathcal{C}}}, \phi_{\mathcal{I}})$, it follows that the run π^p is winning and there exists $f \geq 0$ such that $(v_f, p) \models \phi_f$ and for any $0 \leq i \leq f$, $(v_i, p) \models \phi_{inv}$. Also note that the run $\pi^p_{\downarrow p} = v_0 v_1 \cdots v_f \cdots$ is a run in the game structure $\mathcal{G}$ where the actions are chosen according to the instantiated strategy $\mathcal{S}_{\downarrow p}$. Note that $v_0 \models \phi_{init_{\mathcal{C}_{\downarrow p}}}$, $v_f \models \phi_{f_{\downarrow p}}$, and for any $0 \leq i \leq f$, $v_i \models \phi_{inv_{\downarrow p}}$. Hence we have $\pi^p_{\downarrow p} \models \phi_{\mathcal{I}_{\downarrow p}}$, that is, $\mathcal{S}_{\downarrow p}$ is winning in the game $(\mathcal{G}, \phi_{init_{\mathcal{C}_{\downarrow p}}}, \phi_{\mathcal{I}_{\downarrow p}})$. It is easy to see that $\mathcal{F}_{\downarrow p} \subseteq [\![\phi_{f_{\downarrow p}}]\!]$. Therefore, it follows that for any parameter valuation $p \in \Sigma^a_{\mathcal{P}}$, $\mathcal{C}_{\downarrow p}$ realizes the instantiated interface $\mathcal{I}_{\mathcal{C}_{\downarrow p}}$. It is easy to see that $\phi_{init_{\mathcal{C}}} \rightarrow \phi_{init}$ and $\phi_{f_{\mathcal{C}}} \rightarrow \phi_f$. Note that $\phi_{inv_{\mathcal{C}}} = [\![\phi_{\mathcal{R}}]\!] \subseteq \phi_{inv}$ since $\mathcal{S}$ is a parametric winning strategy in the game $(\mathcal{G}^{\mathcal{P}}, \phi_{init_{\mathcal{C}}}, \phi_{\mathcal{I}})$. Therefore, it follows that $\mathcal{I}_{\mathcal{C}}$ respects $\mathcal{I}$.

Now we show that Algorithm 4.1 is complete, that is, if there exists a parametric controller $\mathcal{C}' = (\mathcal{S}', \mathcal{F}')$ with interface $\mathcal{I}_{\mathcal{C}'} = (\phi_{init_{\mathcal{C}'}}, \phi_{inv_{\mathcal{C}'}}, \phi_{f_{\mathcal{C}'}})$ and a set of admissible parameter values $\Sigma'_{\mathcal{P}} \subseteq \Sigma_{\mathcal{P}}$ satisfying the conditions of Problem 4.1, then Algorithm 4.1 computes such a controller. Let $p \in \Sigma'_{\mathcal{P}}$ be an admissible parameter valuation. Consider any run $\pi^p_{\downarrow p} = v_0 v_1 v_2 \cdots \in \Sigma_{\mathcal{V}}$ in the game structure $\mathcal{G}$ where $v_0 \models \phi_{init_{\mathcal{C}'_{\downarrow p}}}$ and player-2 actions are

chosen according to the instantiated strategy $\mathcal{S}'_{\downarrow p}$. Since $\mathcal{C}'$ realizes the interface $\mathcal{I}_{\mathcal{C}'}$ and $\mathcal{I}_{\mathcal{C}'}$ respects the user-specified interface $\mathcal{I}$, it follows that $\pi^p_{\downarrow p} \models \phi_{\mathcal{I}_{\downarrow p}}$, i.e., the run $\pi^p_{\downarrow p}$ is winning in the game structure $\mathcal{G}$ with respect to $\phi_{\mathcal{I}_{\downarrow p}}$. Consider the run $\pi^p = (v_0, p)(v_1, p)(v_2, p) \cdots \in \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{P}}$ obtained from $\pi^p_{\downarrow p}$ by concatenating the parameter valuations to the states. We have $(v_0, p) \models \phi_{init_{\mathcal{C}'}}$ and $\pi^p \models \phi_{\psi_{\mathcal{C}'}}$ where $\psi_{\mathcal{C}'} = \phi_{inv_{\mathcal{C}'}} \mathcal{U} (\phi_{inv_{\mathcal{C}'}} \wedge \phi_{f_{\mathcal{C}'}})$, and because $\mathcal{I}_{\mathcal{C}'}$ respects $\mathcal{I}$, it follows that $(v_0, p) \models \phi_{init}$ and $\pi^p \models \phi_{\mathcal{I}}$. Note that π^p is a run of parametric game structure $\mathcal{G}^{\mathcal{P}}$ that satisfies the objective $\phi_{\mathcal{I}}$. Since any run π^p in the parametric game structure $\mathcal{G}^{\mathcal{P}}$ starting from $(v_0, p) \models \phi_{init_{\mathcal{C}'}}$ where the actions of player-2 are chosen according to $\mathcal{S}'$ enforces the objective $\phi_{\mathcal{I}}$, it follows that (v_0, p) is a winning state in $\mathcal{G}^{\mathcal{P}}$ with respect to $\phi_{\mathcal{I}}$, and therefore it belongs to the set of winning states $[\![\phi_{\mathcal{W}}]\!]$ computed in line 4 of Algorithm 4.1. Indeed, since $[\![\phi_{\mathcal{W}}]\!]$ is the set of all winning states, it follows that $[\![\phi_{init_{\mathcal{C}'}}]\!] \subseteq [\![\phi_{init_{\mathcal{C}}}]\!]$ and $\Sigma'^a_{\mathcal{P}} \subseteq \Sigma^a_{\mathcal{P}}$. Given that the set of winning states $[\![\phi_{\mathcal{W}}]\!]$ and the set $\phi_{init_{\mathcal{C}}}$ of winning initial states are not empty, Algorithm 4.1 computes a controller $\mathcal{C}$, its interface $\mathcal{I}_{\mathcal{C}}$, and a set of parameter valuations $\Sigma^a_{\mathcal{P}}$ such that $\mathcal{I}_{\mathcal{C}}$ respects $\mathcal{I}$ and any $p \in \Sigma^a_{\mathcal{P}}$ is admissible.

Next we discuss the complexity of Algorithm 4.1. The most computationally expensive parts of Algorithm 4.1 are lines $4, 8$, and 9 that can be computed in $O(|\Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{P}}|)$, i.e., in linear time in the size of the parametric game structure [MPS95]. To see that the algorithm performs only $O(|\Sigma_{\mathcal{V}}|)$ number of symbolic steps, first note that the value of the parameters stay constant over transitions of the parametric game structure $\mathcal{G}^{\mathcal{P}}$. Intuitively, for any parameter valuation $p \in \Sigma_{\mathcal{P}}$, the parametric game structure has a copy of the game structure $\mathcal{G}$, where each copy is independent and there is no transition between different copies (see Figure 4.4 for an example). Furthermore, each copy has the same size as the original game structure $\mathcal{G}$, that is, each copy is of size $|\Sigma_{\mathcal{V}}|$. The symbolic algorithm computes the controllable predecessors of any set of parametric states simultaneously for all parameter valuations over parallel copies in the parametric game structure. Since there is no transition between copies corresponding to each parameter valuations and each copy needs $O(|\Sigma_{\mathcal{V}}|)$ symbolic steps in the worst case, it follows that Algorithm 4.1 performs $O(|\Sigma_{\mathcal{V}}|)$ symbolic steps in the worst case. □

The number of symbolic steps is not the only factor determining the time taken by the symbolic algorithm, however, it is an important measure of difficulty since image and pre-image computations are typically the most expensive operations [BGS06]. Intuitively,

the number of symbolic steps in Algorithm 4.1 is independent of the parameters because the symbolic algorithm for computing the set of winning states can manipulate the parallel copies in the parametric game structure simultaneously, and that each copy is of size $O(|\Sigma_{\mathcal{V}}|)$ (the parametric game structure can be viewed as $|\Sigma_{\mathcal{P}}|$ copies of the original game structure.) As an example, consider the parametric game structure in Figure 4.4. Observe that each copy for each parameter valuation is of the same size of the non-parametric game structure shown in Figure 4.3. Let $\Phi = (x = p)$ be a parametric predicate. It is easy to see that states q_0^0, q_1^1, and q_2^2 in the parametric game structure satisfy Φ. The pre-image of these states (states that can reach them in one step) is the set $\{q_0^1, q_1^0, q_2^1, q_1^2, q_3^2\}$ that is computed by the symbolic algorithm in one step. Note that although the number of symbolic steps in Algorithm 4.1 is independent of the parameters, it does not mean that adding parameters has no additional cost as they may increase the complexity of the symbolic steps. However, transitions over parameters have a special structure that may be utilized for efficient implementation of the symbolic steps.

4.4 Synthesis of Control Strategy with Parametric Controllers

In this section we describe our solution for Problem 5.1 stated in Section 4.2. Our goal is to synthesize a control strategy $\mathcal{S}^{\mathcal{C}}$ such that its induced strategy is winning in the game $(\mathcal{G}, \phi_{init}, \Phi)$. To this end, we first obtain a *control game structure* $\mathcal{G}^{\mathcal{C}}$ using the set of controller interfaces $\Gamma_{\mathcal{I}_{\mathcal{C}}}$. Intuitively, $\mathcal{G}^{\mathcal{C}}$ models what controllers and parameter valuations the system can choose at any state, possible states that may be visited while the selected controller is active, and potential final states that may be reached once the controller is done. From the standpoint of the composer, each instantiated controller that becomes active goes through three steps: *initialization* (the controller starts its execution from a valid initial state), *execution* (the state of the system evolves according to the controller), and *termination* (the controller enters a final state and returns the control).

Formally, let $\gamma_{\mathcal{C}} \notin \mathcal{V}$ defined over the domain $\Sigma_{\gamma_{\mathcal{C}}} = [1..n]$ be a variable representing the controllers, i.e., $\gamma_{\mathcal{C}} = i$ corresponds to the controller $\mathcal{C}_i \in \Gamma_{\mathcal{C}}$ for $i = 1, \cdots, n$. Let $t_c \notin \mathcal{V}$ defined over $\Sigma_{t_c} = \{1, 2\}$ be a variable indicating which player's turn it is in the control game structure. Moreover, let $t_e \notin \mathcal{V}$ defined over $\Sigma_{t_e} = \{1, 2\}$ be an additional variable

that player-1 uses to distinguish a controller's possible initial states from *intermediate* states that may be visited during the execution of the controller. A control game structure $\mathcal{G}^{\mathcal{C}}$ is a tuple $(\mathcal{V}^{\mathcal{C}}, \Lambda^{\mathcal{C}}, \tau^{\mathcal{C}})$ where $\mathcal{V}^{\mathcal{C}} = \mathcal{V} \cup \{t_c, t_e, \gamma_{\mathcal{C}}\} \cup \mathcal{P}$ is a set of variables defined over the domain $\Sigma_{\mathcal{V}^{\mathcal{C}}} = \Sigma_{\mathcal{V}} \times \Sigma_{t_c} \times \Sigma_{t_e} \times \Sigma_{\gamma_{\mathcal{C}}} \times \Sigma_{\mathcal{P}}$, $\Lambda^{\mathcal{C}} = \Sigma_{\mathcal{P}'} \times \Sigma_{\gamma'_{\mathcal{C}}}$ is a set of actions, and $\tau^{\mathcal{C}}$ is symbolically defined as $\tau^{\mathcal{C}} = \bigvee_{i=1}^{n} (\tau_s^{\mathcal{C}_i} \vee \tau_{e1}^{\mathcal{C}_i} \vee \tau_{e2}^{\mathcal{C}_i})$ with

$$
\begin{aligned}
\tau_s^{\mathcal{C}_i} &:= t_c = 2 \wedge same(\mathcal{V}, \mathcal{V}') \wedge t'_c = 1 \wedge t'_e = 1 \wedge \gamma'_c = i \wedge \phi'_{init_{\mathcal{C}_i}}, \\
\tau_{e1}^{\mathcal{C}_i} &:= t_c = 1 \wedge \gamma_c = i \wedge t_e = 1 \wedge t'_c = 1 \wedge t'_e = 2 \wedge \phi'_{inv_{\mathcal{C}_i}} \wedge \\
&\qquad same(\mathcal{P}, \mathcal{P}') \wedge same(\gamma_c, \gamma'_c), \text{ and} \\
\tau_{e2}^{\mathcal{C}_i} &:= t_c = 1 \wedge t_e = 2 \wedge \gamma_c = i \wedge t'_c = 2 \wedge \phi'_{f_{\mathcal{C}_i}} \wedge \\
&\qquad same(\mathcal{P}, \mathcal{P}') \wedge same(\gamma_c, \gamma'_c).
\end{aligned}
$$

In the above predicates, $\gamma'_{\mathcal{C}}$ is a primed copy of $\gamma_{\mathcal{C}}$, and ϕ' is obtained by replacing variables in ϕ by their primed copies. Note that the primed copies of the parameters and $\gamma_{\mathcal{C}}$ encode the actions $\Lambda^{\mathcal{C}}$ of the control game structure to indicate that the composer (player-2 in $\mathcal{G}^{\mathcal{C}}$) selects the parameter valuation and the parametric controller when it is her turn, and also to avoid introducing additional variables. We denote by $\Sigma^i_{\mathcal{V}^{\mathcal{C}}} = \left\{ v^{\mathcal{C}} \in \Sigma_{\mathcal{V}^{\mathcal{C}}} \mid v^{\mathcal{C}}_{|t_c} = i \right\}$ the set of player-i states in the control game structure for $i = 1, 2$.

At any player-2 state in the control game structure, the composer must choose a controller $\mathcal{C} \in \Gamma_{\mathcal{C}}$ and an admissible parameter valuation $p \in \Sigma_{\mathcal{P}}$, if one exists. Furthermore, the composer must ensure that the selected controller starts from a valid initial state, i.e., the state where the instantiated controller $\mathcal{C}_{\downarrow p}$ receives the control satisfies the predicate $\phi_{init_{\mathcal{C}_{\downarrow p}}}$. This is captured in the predicate $\tau_s^{\mathcal{C}_i}$ of $\tau^{\mathcal{C}}$ for each controller $\mathcal{C}_i$. According to $\tau_s^{\mathcal{C}_i}$ at any state $(v, t_c = 2, t_e, \gamma_{\mathcal{C}}, p) \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$, the controller $\mathcal{C}_i$ can be chosen by selecting $\gamma'_{\mathcal{C}} = i$ if there exists a parameter valuation $p' \in \Sigma_{\mathcal{P}'}$ such that the initial condition of the controller is satisfied, i.e., $(v', p') \models \phi'_{init_{\mathcal{C}_i}}$ where v' is obtained by replacing variables in v by their primed copies. $t'_c = 1$ means that it is player-1 state in the next turn, and $t'_e = 1$ means that player-1 states in the next turn satisfy the initial condition of the controller. Intuitively, each predicate $\tau_s^{\mathcal{C}_i}$ for $i = 1..n$ models initialization of a controller $\mathcal{C}_i$.

Once a controller $\mathcal{C}_i$ and a parameter valuation $p \in \Sigma_{\mathcal{P}}$ are selected by the composer, the control is transferred to the instantiated controller $\mathcal{C}_{i_{\downarrow p}}$, and the controller and parameter valuation are fixed until the control is returned to the composer. This is captured in $\tau_{e1}^{\mathcal{C}_i}$

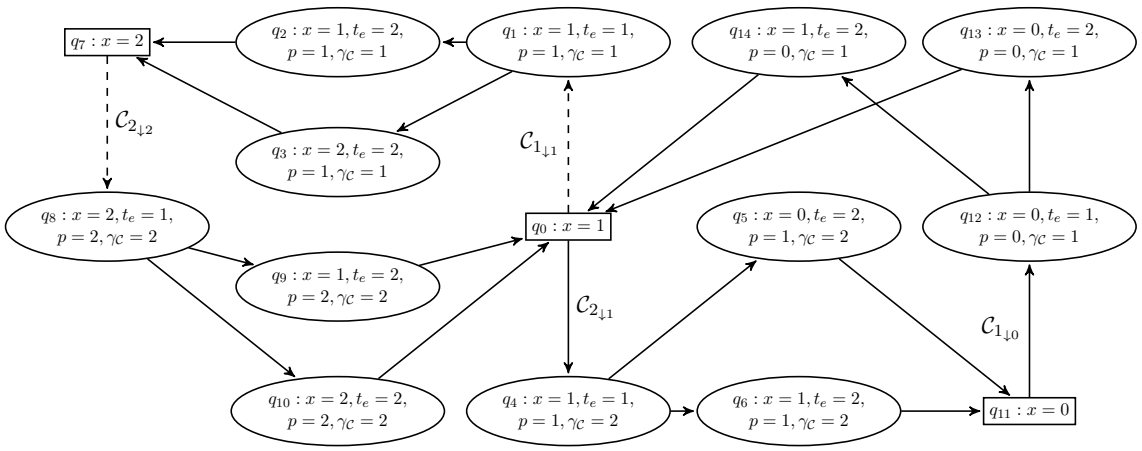


Figure 4.5: Control game structure for Example 4.2 where player-2 states are grouped together for a compact representation. Outgoing edges from player-2 states are labeled by an instantiated controller that the composer can choose at those states. A Control strategy for objective $\Phi = \Box(x \neq 2) \wedge \Diamond(x = 1)$ is to choose solid edges at player-2 states.

and $\tau_{e_2}^{\mathcal{C}_i}$ by $same(\mathcal{P}, \mathcal{P}') \wedge same(\gamma_{\mathcal{C}}, \gamma'_{\mathcal{C}})$. Player-1 states with $t_e = 1$ ($t_e = 2$) and $\gamma_{\mathcal{C}_i} = i$ in $\mathcal{G}^{\mathcal{C}}$ represent initial (intermediate) states where the predicate $\phi_{init_{\mathcal{C}_{\downarrow p}}}$ ($\phi_{inv_{\mathcal{C}_{\downarrow p}}}$, respectively) of the instantiated controller interface $\mathcal{I}_{\mathcal{C}_{i_{\downarrow p}}}$ is satisfied. Intuitively, each predicate $\tau_{e_1}^{\mathcal{C}_i}$ captures transitions from controller's initial state to its intermediate states (representing the execution of the controller), and $\tau_{e_2}^{\mathcal{C}_i}$ shows the transition from intermediate states to final states (modeling termination) where the controller has reached a final state and control is returned to the composer. We illustrate the ideas with a simple example.

Example 4.2. *Let $x \in [0..2]$ be a variable, and $p \in [0..2]$ be a parameter. Consider two controllers $\mathcal{C}_1$ and $\mathcal{C}_2$ with controller interfaces $\mathcal{I}_{\mathcal{C}_1} = (\phi_{init_{\mathcal{C}_1}}, \phi_{inv_{\mathcal{C}_1}}, \phi_{f_{\mathcal{C}_1}})$ and $\mathcal{I}_{\mathcal{C}_2} = (\phi_{init_{\mathcal{C}_2}}, \phi_{inv_{\mathcal{C}_2}}, \phi_{f_{\mathcal{C}_2}})$ defined as follows:*

- $\phi_{init_{\mathcal{C}_1}} = (\phi_{\mathcal{P}_1} \wedge x = p)$,
- $\phi_{f_{\mathcal{C}_1}} = (\phi_{\mathcal{P}_1} \wedge (x = p + 1))$,
- $\phi_{inv_{\mathcal{C}_1}} = \phi_{init_1} \vee \phi_{f_{\mathcal{C}_1}}$,
- $\phi_{init_{\mathcal{C}_2}} = (\phi_{\mathcal{P}_2} \wedge x = p)$,
- $\phi_{f_{\mathcal{C}_2}} = (\phi_{\mathcal{P}_2} \wedge (x = p - 1))$, *and*
- $\phi_{inv_{\mathcal{C}_2}} = \phi_{init_2} \vee \phi_{f_{\mathcal{C}_2}}$.

where $\phi_{\mathcal{P}_1} = (0 \leq p \leq 1)$ and $\phi_{\mathcal{P}_2} = (1 \leq p \leq 2)$. Intuitively, $\mathcal{C}_1$ eventually increments the value of x by 1, while $\mathcal{C}_2$ eventually decrements it by 1. Furthermore, $\phi_{inv_{\mathcal{C}_i}} = \phi_{init_i} \vee \phi_{f_{\mathcal{C}_i}}$,

for $i = 1, 2$, indicates that the set of states that are possibly visited during execution of controller $\mathcal{C}_i$ is the union of initial states and final states. Figure 4.5 shows the control game structure for this example where player-2 (player-1) states are depicted by boxes (ovals, respectively) and player-2 states are grouped together based on their valuations over x for a compact representation. Each node of the graph in Figure 4.5 is labeled with a name q_j and a set of predicates that hold in those states. For example, node q_0 represents all player-2 states in the control game structure for which $x = 1$. Nodes q_7 and q_{11} can be interpreted in a similar way. Outgoing edges from player-2 states are labeled by an instantiated controller that the composer can select at those states, e.g., at q_0, the composer can select either the instantiated controller $\mathcal{C}_{1_{\downarrow 1}}$ (corresponding to the action $(p' = 1, \gamma'_{\mathcal{C}} = 1)$,) or the instantiated controller $\mathcal{C}_{2_{\downarrow 1}}$. If the composer chooses $\mathcal{C}_{2_{\downarrow}1}$, then the control of the system is transferred to $\mathcal{C}_{2_{\downarrow}1}$, and q_4 is visited next in the control game structure. Note that the controller and parameter valuations are selected by the composer and they do not change in player-1 states. Once the controller is initialized, any intermediate state that satisfies the invariant of the instantiated controller can be visited in the control game structure (nodes q_5 and q_6 in Figure 4.5,) and at the next step, a final state of the instantiated controller is visited (represented by q_{11}), indicating the execution of the controller is over and the composer must decide the next action.

Once the control game structure is obtained, we solve the *control game* $(\mathcal{G}^{\mathcal{C}}, \phi^{\mathcal{C}}_{init}, \Phi)$ where $\phi^{\mathcal{C}}_{init} = (t_c = 2 \wedge \phi_{init})$, i.e., the control game starts from a player-2 state so that the composer can initially select a controller and a parameter valuations. If player-2 has a winning strategy in the control game, we synthesize a winning strategy $\mathcal{S}^{\Phi}$ of special form that only depends on the valuation over variables $\mathcal{V}$ and then extract a control strategy $S^{\mathcal{C}}$ from it. Formally, let $\Upsilon = \mathcal{V}^{\mathcal{C}} \backslash \mathcal{V}$ be the set of parameters and additional variables introduced for the control game structure. We say a player-2 strategy $\mathcal{S}^{\Phi}$ in the control game structure is Υ-*independent* if there exists a partial function $f^{\mathcal{C}} : \Sigma_{\mathcal{V}} \to \Lambda^{\mathcal{C}}$ such that for any player-2 state $v^{\mathcal{C}} = (v, 2, i, j, p) \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$, $\mathcal{S}^{\Phi}(v, 2, i, j, p) = f^{\mathcal{C}}(v)$. Intuitively, it means that $\mathcal{S}^{\Phi}$ only depends on the valuation over variables $\mathcal{V}$. The following theorem states that if player-2 can win the control game, then a Υ-independent winning strategy can be synthesized.

Theorem 4.2. *If the control game is realizable, then there exists a Υ-independent winning strategy for player-2.*

Proof. Let $\mathcal{W}^{\mathcal{C}}$ be the set of winning states in the control game structure $(\mathcal{G}^{\mathcal{C}}, \Phi^{\mathcal{C}}_{init}, \Phi)$. The following lemma shows that if there exists a player-2 state $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with valuation $v \in \Sigma_{\mathcal{V}}$ over variables $\mathcal{V}$, i.e., $v^{\mathcal{C}}_{|\mathcal{V}} = v$, such that $v^{\mathcal{C}}$ is winning, i.e., $v^{\mathcal{C}} \in \mathcal{W}^{\mathcal{C}}$, then for all $w^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $w^{\mathcal{C}}_{|\mathcal{V}} = v$, we have $w^{\mathcal{C}} \in \mathcal{W}^{\mathcal{C}}$, i.e., any player-2 state with the same valuation $v \in \Sigma_{\mathcal{V}}$ over variables $\mathcal{V}$ is also winning. Intuitively, it means that the existence of a winning strategy at a player-2 state in the control game structure only depends on its valuation over the variables $\mathcal{V}$.

Lemma 4.1. *There exists $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $v^{\mathcal{C}}_{|\mathcal{V}} = v$ such that $v^{\mathcal{C}} \in \mathcal{W}^{\mathcal{C}}$ if and only if for all $w^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $w^{\mathcal{C}}_{|\mathcal{V}} = v$, we have $w^{\mathcal{C}} \in \mathcal{W}^{\mathcal{C}}$.*

Proof. We prove one direction. The other direction is trivial. Let $v^{\mathcal{C}} = (v, 2, j, \gamma, p) \in \Sigma_{\mathcal{V}} \times \{2\} \times \Sigma_{t_e} \times \Sigma_{\gamma_{\mathcal{C}}} \times \Sigma_{\mathcal{P}} = \Sigma^2_{\mathcal{V}_{\mathcal{C}}}$ be a player-2 state with $v^{\mathcal{C}}_{|\mathcal{V}} = v \in \Sigma_{\mathcal{V}}$ such that $v^{\mathcal{C}} \in \mathcal{W}^{\mathcal{C}}$. Then by definition of winning states, there must exist an action $\ell = (p^*, i) \in \Sigma_{\mathcal{P}'} \times \Sigma_{\gamma'_{\mathcal{C}}}$ such that all ℓ-successor states $v'^{\mathcal{C}} \in Succ(v^{\mathcal{C}}, \ell)$ are winning, i.e., $v'^{\mathcal{C}} \in \mathcal{W}^{\mathcal{C}}$. It follows that for any $v'^{\mathcal{C}} \in Succ(v^{\mathcal{C}}, \ell)$ we have $(v^{\mathcal{C}}, \ell, v'^{\mathcal{C}}) \models \tau^{\mathcal{C}_i}_s$. Note that in the control game structure, there exists an outgoing transition from a player-2 state if and only if there exist a truth assignment to the variables $\mathcal{V}^{\mathcal{C}}$ and their primed copies such that $t^{\mathcal{C}_k}_s$ is satisfied for some $1 \leq k \leq n$. Furthermore, note that the predicate $\tau^{\mathcal{C}_i}_s$ does not depend on the current value of the variables $t_e, \gamma_{\mathcal{C}}$, and $\mathcal{P}$. Thus, for all $(l, c, p_2) \in \Sigma_{t_e} \times \Sigma_{\gamma_{\mathcal{C}}} \times \Sigma_{\mathcal{P}}$ and for any state $w^{\mathcal{C}} = (v, 2, l, c, p_2) \in \Sigma^2_{\mathcal{V}_{\mathcal{C}}}$, and for all $v'^{\mathcal{C}} \in Succ(v^{\mathcal{C}}, \ell)$, $(w^{\mathcal{C}}, \ell, v'^{\mathcal{C}}) \models \tau^{\mathcal{C}_i}_s$ and it follows that $w^{\mathcal{C}}$ is also a winning state, i.e., $w^{\mathcal{C}} \in \mathcal{W}^{\mathcal{C}}$. □

Let Φ be a safety objective, and $\phi_{\mathcal{W}}$ be a predicate specifying the set of winning states in the game structure $\mathcal{G}^{\mathcal{C}}$ with respect to Φ. Let $\tau^{\mathcal{W}} := (t_c = 2 \wedge \tau^{\mathcal{C}} \wedge \phi_{\mathcal{W}} \wedge \phi'_{\mathcal{W}}) = (\bigvee^n_{i=1} \tau^{\mathcal{C}_i}_s) \wedge \phi_{\mathcal{W}} \wedge \phi'_{\mathcal{W}}$ characterize the set of outgoing transitions from player-2 states that keep the state of the game in the set of winning states, where $\phi'_{\mathcal{W}}$ is obtained from $\phi_{\mathcal{W}}$ by replacing its variables by their primed copies. We construct a Υ-independent strategy $\mathcal{S}^{\Phi}$ with a corresponding partial function $f^{\mathcal{C}} : \Sigma_{\mathcal{V}} \to \Lambda^{\mathcal{C}}$ from $\tau^{\mathcal{W}}$ as follows. Let $(t^*_e, \gamma^*_{\mathcal{C}}, p^*) \in \Sigma_{t_e} \times \Sigma_{\gamma_{\mathcal{C}}} \times \Sigma_{\mathcal{P}}$ be arbitrarily chosen values for variables t_e, $\gamma_{\mathcal{C}}$, and $\mathcal{P}$. For any $v^{\mathcal{C}} = (v, 2, l, c, p) \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ such that $v^{\mathcal{C}} \models \phi_{\mathcal{W}}$, i.e., $v^{\mathcal{C}}$ is a winning player-2 state, we let $f^{\mathcal{C}}(v) = (p', c') \in \Sigma_{\mathcal{P}'} \times \Sigma_{\gamma'_{\mathcal{C}}}$ and $\mathcal{S}^{\Phi}(v^{\mathcal{C}}) = f^{\mathcal{C}}(v^{\mathcal{C}}_{|\mathcal{V}})$ where (p', c') is an arbitrary truth assignment to $\mathcal{P}'$ and $\gamma'_{\mathcal{C}}$ that is *consistent* with $\tau^{\mathcal{W}}$, that is, when variables $\mathcal{V}, t_c, t_e, \gamma_{\mathcal{C}}, \mathcal{P}, \gamma'_{\mathcal{C}}$, and $\mathcal{P}'$ are replaced by valuations $v, 2, t^*_e, \gamma^*_{\mathcal{C}}, p^*, c'$, and p' in $\tau_{\mathcal{W}}$, respectively, the resulting formula is satisfiable.

Intuitively, it means that $(p', \mathcal{C}_{c'})$ is a controller and parameter valuation that is consistent with $\tau^{\mathcal{W}}$ and leads to a next state that is winning. It follows from Lemma 4.1 that, if $v^{\mathcal{C}}$ is winning, then any state $w^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ that has the same valuation over variables $\mathcal{V}$, i.e., $v^{\mathcal{C}}_{\restriction \mathcal{V}} = w^{\mathcal{C}}_{\restriction \mathcal{V}}$, is also winning. It then follows that the constructed strategy is a Υ-independent winning strategy.

Now let Φ be a reachability objective. The least fixed-point algorithm for computing the set of winning states for reachability objective computes a sequence $\mathcal{W}_0 \subseteq \mathcal{W}_1 \subset \cdots \subseteq \mathcal{W}_d$ for $d \geq 0$ until a fixed-point is reached where $\mathcal{W}_i \subseteq \Sigma_{\mathcal{V}^{\mathcal{C}}}$ and $\mathcal{W}_d = \mathcal{W}^{\mathcal{C}}$ is the fixed-point that characterizes the set of winnig states. Let $\mathcal{W}_i^- = \mathcal{W}_i \setminus \bigcup_{j=0}^{i-1} \mathcal{W}_j$ be the set of winning states that are discovered for the first time at i-th step, for $0 \leq i \leq d$. A winning strategy $\mathcal{S}^{\Phi}_{\#}$ enforcing the reachability objective chooses the actions such that any player-2 state in layer i, i.e., $v^{\mathcal{C}} \in \mathcal{W}_i^-$ will reach a state $v'^{\mathcal{C}}$ in lower layers, i.e., $v'^{\mathcal{C}} \in \bigcup_{j=0}^{i-1} \mathcal{W}_j$. We will construct a Υ-independent strategy $\mathcal{S}^{\Phi}$ with a corresponding partial function $f^{\mathcal{C}} : \Sigma_{\mathcal{V}} \to \Lambda^{\mathcal{C}}$ from the winning strategy $\mathcal{S}^{\Phi}_{\#}$. The following lemma states that if a player-2 winning state $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with valuation v over variables $\mathcal{V}$, i.e., $v^{\mathcal{C}}_{\restriction \mathcal{V}} = v \in \Sigma_{\mathcal{V}}$ belongs to winning layer $\mathcal{W}_i^-$ for some $0 \leq i \leq d$ then all player-2 states $w^{\mathcal{C}}$ with the same valuation v over $\mathcal{V}$ belong to the winning layer $\mathcal{W}_i^-$.

Lemma 4.2. *Let $\mathcal{W}_0, \mathcal{W}_1, \cdots, \mathcal{W}_d$ for $d \geq 0$ be a sequence of sets of states obtained during the fixed-point computation for reachability objectives such that $\mathcal{W}_d$ is the fixed point, and characterizes the set of winning states, i.e., $\mathcal{W}^{\mathcal{C}} = \mathcal{W}_d$. Let $\mathcal{W}_i^- = \mathcal{W}_i \setminus \bigcup_{j=0}^{i-1} \mathcal{W}_j$ be the set of winning states that are discovered for the first time at i-th step, for $0 \leq i \leq d$. There exists a player-2 state $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $v^{\mathcal{C}}_{\restriction \mathcal{V}} = v \in \Sigma_{\mathcal{V}}$ such that $v^{\mathcal{C}} \in \mathcal{W}_i^-$ for some $0 \leq i \leq d$ if and only if for all $w^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $w^{\mathcal{C}}_{\restriction \mathcal{V}} = v$, $w^{\mathcal{C}} \in \mathcal{W}_i^-$ for some $0 \leq i \leq d$.*

Proof. We prove the lemma for a reachability objective Φ where the fixed-point computation starts from the set $\mathcal{W}_0 = \emptyset$. We prove it for one direction. The other direction is trivial. We show by induction that for any $i \geq 0$, if for a player-2 state $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $v^{\mathcal{C}}_{\restriction \mathcal{V}} = v \in \Sigma_{\mathcal{V}}$ we have $v^{\mathcal{C}} \in \mathcal{W}_i^-$, then for all $w^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $w^{\mathcal{C}}_{\restriction \mathcal{V}} = v$, $w^{\mathcal{C}} \in \mathcal{W}_i$. It holds trivially for $\mathcal{W}_0 = \emptyset$. Consider $\mathcal{W}_i$ for $0 < i < d$ (note that for any $i > d$, $\mathcal{W}_i = \mathcal{W}_d$ and therefore, $\mathcal{W}_i^- = \emptyset$). Let $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with $v^{\mathcal{C}}_{\restriction \mathcal{V}} = v \in \Sigma_{\mathcal{V}}$ be a player-2 state with valuation v over the variables $\mathcal{V}$ such that $v^{\mathcal{C}} \in win^-_{i+1}$. Then by definition of winning states, there must exist an action $\ell = (p^*, c^*) \in \Sigma_{\mathcal{P}'} \times \Sigma_{\gamma'_{\mathcal{C}}}$ such that all successor states $v'^{\mathcal{C}} \in Succ(v^{\mathcal{C}}, \ell)$ are winning, i.e.,

$v'^{\mathcal{C}} \in \bigcup_{j=0}^{i-1} \mathcal{W}_j$. It follows that for any $v'^{\mathcal{C}} \in Succ(v^{\mathcal{C}}, \ell)$, we have $(v^{\mathcal{C}}, \ell, v'^{\mathcal{C}}) \models \tau_s^{\mathcal{C}_{c^*}}$. Note that there exists a outgoing transition from a player-2 state in the control game structure if and only if there exist a truth assignment to the variables $\mathcal{V}^{\mathcal{C}}$ and their primed copies such that $t_s^{\mathcal{C}_i}$ is satisfied for some $1 \leq i \leq n$. Furthermore, note that the predicate $\tau_s^{\mathcal{C}_i}$ does not depend on the current value of the variables $t_e, \gamma_{\mathcal{C}}$, and $\mathcal{P}$. Thus, for all $(l, c, p_2) \in \Sigma_{t_e} \times \Sigma_{\gamma_{\mathcal{C}}} \times \Sigma_{\mathcal{P}}$ and for any state $w^{\mathcal{C}} = (v, 2, l, c, p_2) \in \Sigma^2_{\mathcal{V}_{\mathcal{C}}}$ and all $v'^{\mathcal{C}} \in Succ(v^{\mathcal{C}}, \ell)$, we have $(w^{\mathcal{C}}, \ell, v'^{\mathcal{C}}) \models \tau_s^{\mathcal{C}_{c^*}}$ and it follows that $w^{\mathcal{C}}$ is also a winning state, i.e., $w^{\mathcal{C}} \in win^-_{i+1}$. □

We construct a Υ-independent strategy $\mathcal{S}^{\Phi}$ with a corresponding partial function $f^{\mathcal{C}} : \Sigma_{\mathcal{V}} \to \Lambda^{\mathcal{C}}$ from the winning strategy $\mathcal{S}^{\Phi}_{\#}$. Let $(t_e^*, \gamma_{\mathcal{C}}^*, p^*) \in \Sigma_{t_e} \times \Sigma_{\gamma_{\mathcal{C}}} \times \Sigma_{\mathcal{P}}$ be arbitrarily chosen values for variables t_e, $\gamma_{\mathcal{C}}$, and $\mathcal{P}$. For any $v^{\mathcal{C}} = (v, 2, i, j, p) \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ such that $\mathcal{S}^{\mathcal{C}}_{\#}(v^{\mathcal{C}})$ is defined, we let $f^{\mathcal{C}}(v) = \mathcal{S}^{\mathcal{C}}_{\#}((v, 2, t_e^*, \gamma_{\mathcal{C}}^*, p^*))$ and $\mathcal{S}^{\Phi}(v^{\mathcal{C}}) = f^{\mathcal{C}}(v^{\mathcal{C}}_{|\mathcal{V}})$. It is easy to see that $\mathcal{S}^{\Phi}$ is Υ-independent. By Lemma 4.2 and since $\mathcal{S}^{\Phi}_{\#}$ is winning, it follows that $\mathcal{S}^{\Phi}$ is also a winning strategy. □

Note that the predicates $\tau_s^{\mathcal{C}_i}$ in the transition relation $\tau^{\mathcal{C}}$ of $\mathcal{G}^{\mathcal{C}}$ do not depend on the variables $t_e, \gamma_{\mathcal{C}}$ or parameters (though depending on their primed copies,) and the value of $t_c = 2$ is fixed. Intuitively, it means the composer can select the next controller and parameter valuations only based on the current valuation over $\mathcal{V}$ and regardless of current values of parameters, t_e and $\gamma_{\mathcal{C}}$. A control strategy $\mathcal{S}^{\mathcal{C}} : \Sigma_{\mathcal{V}} \to \Sigma_{\mathcal{P}} \times \Gamma_{\mathcal{C}}$ is extracted from $\mathcal{S}^{\Phi}$ using the function $f^{\mathcal{C}}$ as follows. For any $v \in \Sigma_{\mathcal{V}}$ such that $f^{\mathcal{C}}(v)$ is defined and $f^{\mathcal{C}}(v) = (p', i) \in \Sigma_{\mathcal{P}'} \times \Sigma_{\gamma'_{\mathcal{C}}}$, we let $\mathcal{S}^{\mathcal{C}}(v) = (p, \mathcal{C}_i)$ where p is obtained by replacing primed copies of parameters in p' by their unprimed versions. Algorithm 4.2 summarizes the steps for computing $\mathcal{S}^{\mathcal{C}}$. The following theorem establishes the correctness and the complexity of Algorithm 4.2.

Theorem 4.3. *Algorithm 4.2 is sound. For reachability and safety objectives, it performs $O(|\Sigma_{\mathcal{V}}|)$ symbolic steps in the worst case.*

Proof. Let $\mathcal{S}^{\mathcal{C}}$ be a control strategy obtained from Υ-independent winning strategy $\mathcal{S}^{\Phi}$. We show that the strategy $\mathcal{S}$ induced by $\mathcal{S}^{\mathcal{C}}$ is winning in the game $(\mathcal{G}, \Phi_{init}, \Phi)$. Intuitively, we show that any run π in the game structure $\mathcal{G}$ when controllers and parameters are chosen according to the control strategy $\mathcal{S}^{\mathcal{C}}$ corresponds to a set $\Pi^{\mathcal{C}}$ of runs in the control game structure that are all winning for the safety and/or reachability objective Φ, and since the

Algorithm 4.2: Control Strategy Synthesis

Input: A predicate ϕ_{init} specifying a set of initial states, a non-parametric safety and reachability objective Φ, a set $\Gamma_{\mathcal{I}_\mathcal{C}}$ of controller interfaces

Output: A control strategy $\mathcal{S}^\mathcal{C}$ s.t. its induced strategy is winning in the game $(\mathcal{G}, \phi_{init}, \Phi)$

1 Obtain the control game structure $\mathcal{G}^\mathcal{C}$ using $\Gamma_{\mathcal{I}_\mathcal{C}}$;
2 $\phi^\mathcal{C}_{init} := t_c = 2 \wedge \phi_{init}$;
3 Synthesize a $\mathcal{V}^\mathcal{C} \backslash \mathcal{V}$-independent winning strategy $\mathcal{S}^\Phi$ by solving the game $(\mathcal{G}^\mathcal{C}, \phi^\mathcal{C}_{init}, \Phi)$;
4 Extract and return a control strategy $\mathcal{S}^\mathcal{C}$ from $\mathcal{S}^\Phi$;

run π can be obtained by "merging" the runs in $\Pi^\mathcal{C}$, it follows that any run π in the game structure $\mathcal{G}$ also satisfies Φ, and is therefore winning. Let $\mathcal{G}[\mathcal{S}]$ be a restriction of the game structure $\mathcal{G}$ where actions of player-2 are restricted to the actions allowed by the strategy $\mathcal{S}$. Similarly, let $\mathcal{G}^\mathcal{C}[\mathcal{S}^\Phi]$ be a restriction of the control game structure where actions of player-2 are chosen according to the strategy $\mathcal{S}^\Phi$. Note that since $\mathcal{S}^\Phi$ is a winning strategy in the control game $(\mathcal{G}^\mathcal{C}, \phi^\mathcal{C}_{init}, \Phi)$, any run of $\mathcal{G}^\mathcal{C}[\mathcal{S}^\mathcal{C}]$ starting from an initial state $v^\mathcal{C} \models \phi^\mathcal{C}_{init}$ is winning, i.e., it satisfies the objective Φ. Let $\pi = v_{f_0} v_{0_1} v_{0_2} \cdots v_{0_{d_0}} v_{f_1} v_{1_1} \cdots v_{1_{d_1}} v_{f_2} \cdots$ be any run of the restricted game $\mathcal{G}[\mathcal{S}]$ where $v_{f_0} \models \phi_{init}$, i.e., π is a run in the game structure $\mathcal{G}$ starting from a valid initial state where player-2 chooses its actions according to the strategy $\mathcal{S}$. Let $(p, \mathcal{C}) = \mathcal{S}^\mathcal{C}(v_{f_i})$ be the controller and parameter valuation chosen by the control strategy at states v_{f_i} for $i \geq 0$. Let $\mathcal{C}^\#_i = \mathcal{C}_{\downarrow} p$ be the corresponding instantiated controller. As the instantiated controller $\mathcal{C}^\#_i$ realizes its corresponding controller interface, it follows that for $i \geq 0$ we have $v_{f_i} \models \phi_{init_{\mathcal{C}^\#_i}}$, $v_{i_1} \cdots v_{i_{d_i}} \in \Sigma^a_\mathcal{V}$ and for $1 \leq j \leq d_i$, $v_{i_j} \models \phi_{inv_{\mathcal{C}^\#_i}}$, and $v_{f_k} \models \phi_{f_{\mathcal{C}^\#_i}}$ for $k \geq 1$. Intuitively, states $v_{f_i} \in \Sigma_\mathcal{V}$ for $i \geq 0$ are states where a controller and a parameter valuation is chosen by the composer. The selected controller and parameter valuation must be such that v_{f_i} is a valid initial state for the controller. Besides, v_{f_i} for $i > 0$ is also a final state for the controller that relinquishes control, and thus v_{f_i} must also be a final state for the relinquishing controller. The intermediate states $v_{i_1} \cdots v_{i_{d_i}} \in \Sigma^a_\mathcal{V}$ for $i \geq 0$ correspond to set of states that belong to invariant set of the controller and can be visited while the controller guides the execution of the game. Note that the intermediate states may be empty, for example when there is a transition between an initial state and a final state of the selected controller. The run π in the game structure $\mathcal{G}[\mathcal{S}]$ corresponds to a *set* of runs $\Pi^\mathcal{C} = \{v^\mathcal{C}_{f_{0_0}} v^\mathcal{C}_{f_{0_1}} v^\mathcal{C}_{0_1} v^\mathcal{C}_{f_{1_0}} v^\mathcal{C}_{f_{1_1}} v^\mathcal{C}_{1_1} v^\mathcal{C}_{f_{2_0}} \cdots \mid v^\mathcal{C}_{f_{i_{0|\mathcal{V}}}} = v_{f_i}, v^\mathcal{C}_{f_{i_{1|\mathcal{V}}}} = v_{f_i}, v^\mathcal{C}_{i_{k|\mathcal{V}}} \in V^i =$

$\{v_{f_i}\} \cup \bigcup_{j=1}^{d_i} v_{i_j} \cup \{v_{f_{i+1}}\}\}$. Let $\Pi^{\mathcal{C}}_{|\mathcal{V}}$ be a set obtained by projecting the runs in $\Pi^{\mathcal{C}}$ into the set of variables $\mathcal{V}$. Let $\pi^{\mathcal{C}}$ be any run in the set $\Pi^{\mathcal{C}}$. Let $\pi^{\mathcal{C}}_{|\mathcal{V}}$ be the projection of $\pi^{\mathcal{C}}$ into variables $\mathcal{V}$. It is easy to see that $\pi^{\mathcal{C}}_{|\mathcal{V}}$ is of the form $v_{f_0} v_{f_0} w_0 v_{f_1} v_{f_1} w_1 v_{f_2} \cdots$ where $w_i \in V^i = \{v_{f_i}\} \cup \bigcup_{j=1}^{d_i} v_{i_j} \cup \{v_{f_{i+1}}\}$. Note that since $\mathcal{S}^{\Phi}$ is a winning strategy, any run $\pi^{\mathcal{C}} \in \Pi^{\mathcal{C}}$ satisfies the objective Φ. Also since Φ is a predicate over $\Sigma_{\mathcal{V}}$, the projection of $\pi^{\mathcal{C}}$ into $\mathcal{V}$ also satisfies Φ, i.e., $\pi^{\mathcal{C}}_{|\mathcal{V}} \models \Phi$. Note that if a sequence $\sigma = s_0 s_1 \cdots$ of states $s_i \in \Sigma_{\mathcal{V}}$ satisfies a safety objective $\Box(\phi_{\Box})$ and/or a reachability objective $\Diamond(\phi_{\Diamond})$, where $\phi_{\Box}$ and $\phi_{\Diamond}$ are predicates over $\Sigma_{\mathcal{V}}$, then any state $v \in \Sigma_{\mathcal{V}}$ that is safe, i.e., $v \models \phi_{\Box}$ can be placed at any position in σ and the resulting sequence $\sigma' = s_0 s_1 \cdots s_i v s_{i+1} \cdots$ still satisfies the safety and reachability objectives. Also if there exists $i, j \geq 0, i \neq j$ such that $s_i = s_j$, then the sequence $\sigma' = s_0 s_1 \cdots s_i \cdots s_{j-1} s_{j+1} \cdots$ obtained by removing the repetitive state still satisfies the safety and reachability objective. The run π in the game structure $\mathcal{G}$ can be obtained by choosing a sequence $\sigma \in \Pi^{\mathcal{C}}_{|\mathcal{V}}$ and adding states from other sequences in $\Pi^{\mathcal{C}}_{|\mathcal{V}}$ and removing repetitive states, and since all sequences $\sigma \in \Pi^{\mathcal{C}}_{|\mathcal{V}}$ satisfies the safety and/or reachability objective Φ, it follows that π also satisfies Φ. Thus any run in the game $\mathcal{G}[\mathcal{S}]$ is winning and $\mathcal{S}$ is a winning strategy in the game $(\mathcal{G}, \Phi_{init}, \Phi)$.

Complexity. First note that the state space of the control game structure is of size $|\Sigma_{\mathcal{V}^{\mathcal{C}}}|$, and a winning strategy for safety and reachability objectives can be computed in linear time in size of the control game structure, that is, in $O(|\Sigma^{\mathcal{C}}_{\mathcal{V}}|)$. However, the control game structure has a special form that lead to a tighter bound $O(|\Sigma_{\mathcal{V}}|)$ on the number of symbolic steps required to compute the set of winning states for safety or reachability objectives. Intuitively, any infinite run in the control game structure starting from a player-2 state, moves to a player-1 state (an initial state where $t_e = 1$), then moves to another player-1 state (an intermediate state where $t_e = 2$), and finally moves to a player-2 state where the next controller and parameter valuation is chosen and this pattern repeats. That is, along any infinite run in the control game structure starting from a player-2 state, player-1 states are "sandwiched" between player-2 states. Another observation is that during the fixed point computation of the set of winning states, if a player-2 state $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ with valuation v over variables $\mathcal{V}$ is added (or removed) from the current approximation of the set of winning states, then *all* player-2 states with the same valuation over variables $\mathcal{V}$ is added (or removed, respectively) from the set of winning states. This is shown in Lemma 4.2 in proof of Theorem 4.2 for reachability objectives (the proof for safety objectives is similar)

and is due to the fact that $\tau_s^{\mathcal{C}_i}$ in the transition relation $\tau^{\mathcal{C}}$ of the control game structure does not depend on the current value of variables $t_e, \gamma_{\mathcal{C}}$, and $\mathcal{P}$. Finally, let $v^{\mathcal{C}}$ be a player-2 state that belongs to the current set $\mathcal{W}_i \subseteq \Sigma_{\mathcal{V}^{\mathcal{C}}}$ during the fixed point computation of the set of winning states. Recall that in any infinite run of the control game structure any player-2 state is followed by two player-1 state, which is then followed by a player-2 state. Therefore, with at most three symbolic steps from the current approximation of the set of winning state, either a player-2 state $w^{\mathcal{C}}$ with valuation $w \in \Sigma_{\mathcal{V}}$ over variables $\mathcal{V}$ is added to (or removed from) the current approximation, or a fixed point has already reached. In other words, in every three symbolic steps, either a player-2 state is added to (or removed from) the set of the winning states, or a fixed point is reached. Formally,

Lemma 4.3. *Let $\mathcal{W}_0, \mathcal{W}_1, \cdots$ be the sequence of approximations of the set of winning states in the control game structure $\mathcal{G}^{\mathcal{C}}$ with respect to a reachability objective Φ computed by the fixed point algorithm. Assume $d \geq 0$ is the position where the algorithms has reached a fixed point, i.e., $\mathcal{W}_d$ is the set of winning states and for all $j \geq d$, $\mathcal{W}_j = \mathcal{W}_d$. For any $i \geq 0$, either there exists a player-2 state $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$ such that $v^{\mathcal{C}} \notin \mathcal{W}_i$ and $v^{\mathcal{C}} \in \mathcal{W}_{i+3}$, or $\mathcal{W}_{i+3} = \mathcal{W}_{i+2}$ is a fixpoint.*

Proof. Consider the sets $\mathcal{W}_i, \mathcal{W}_{i+1}, \mathcal{W}_{i+2}, \mathcal{W}_{i+3}$ for any $i \geq 0$. Assume for the sake of contradiction that for all player-2 states $v^{\mathcal{C}} \in \Sigma^2_{\mathcal{V}^{\mathcal{C}}}$, if $v^{\mathcal{C}} \in \mathcal{W}_{i+3}$ then $v^{\mathcal{C}} \in \mathcal{W}_i$ and $\mathcal{W}_{i+2} \neq \mathcal{W}_{i+3}$. Therefore, there must exist a player-1 state $w^{\mathcal{C}} \in \Sigma^1_{\mathcal{V}^{\mathcal{C}}}$ such that $w^{\mathcal{C}} \in \mathcal{W}_{i+1}$ but $w^{\mathcal{C}} \notin \mathcal{W}_i$, since otherwise $\mathcal{W}_i = \mathcal{W}_{i+1} = \mathcal{W}_{i+2} = \mathcal{W}_{i+3}$ and we have reached a fixed point. Let $V_{i+1}^{t_e=j} \subseteq \mathcal{W}_{i+1} \backslash \mathcal{W}_i$ be the set of player-1 states with valuation $j \in \{1, 2\}$ over the variable t_e such that they belong to the set $\mathcal{W}_{i+1}$ but not to $\mathcal{W}_i$. Note that at least one of the sets $V_{i+1}^{t_e=1}$ or $V_{i+1}^{t_e=2}$ must be non-empty since otherwise $\mathcal{W}_{i+1}$ is a fixpoint. Consider the set $CPre(V_{i+1}^{t_e=1})$ of controllable predecessors of the player-1 states $v^{\mathcal{C}} \in V_{i+1}^{t_e=1}$ with $v^{\mathcal{C}}_{|t_e} = 1$. Since all predecessors of player-1 states with $t_e = 1$ in the control game structure must be a player-2 state, by our assumption it follows that $\mathcal{W}_{i+2}^{-} \cap CPre(V_{i+1}^{t_e=1}) = \emptyset$ where $\mathcal{W}_{i+2}^{-} = \mathcal{W}_{i+2} \backslash \mathcal{W}_{i+1}$, i.e., there is no winning predecessor in $\mathcal{W}_{i+2}$ for any of the states in $V_{i+1}^{t_e=1}$. Now consider the set $CPre(V_{i+1}^{t_e=2})$ of predecessors of the player-1 states $V_{i+1}^{t_e=2}$. Since all predecessors of player-1 states with $t_e = 2$ in the control game structure must be a player-1 state with $t_e = 1$, by our assumption it follows that $\mathcal{W}_{i+2}^{-} \cap Pre(V_{i+1}^{t_e=2}) = V_{i+2}^{t_e=2}$, where $V_{i+2}^{t_e=2} \subseteq \Sigma^1_{\mathcal{V}^{\mathcal{C}}}$ is a (possibly empty) set of player-1 states with $t_e = 1$. If $V_{i+2}^{t_e=2}$ is empty,

then no new state is added to $\mathcal{W}_{i+2}$, i.e., $\mathcal{W}_{i+1} = \mathcal{W}_{i+2}$ and we have $\mathcal{W}_{i+3} = \mathcal{W}_{i+2}$. Assume $V_{i+2}^{t_e=2}$ is not empty. Then all the predecessors $CPre(V_{i+2}^{t_e=1})$ of the set $V_{i+2}^{t_e=1}$ are player-2 states, and by our assumption it follows that $\mathcal{W}_{i+3}^{-} \cap Pre(V_{i+2}^{t_e=1}) = \emptyset$. That is, no new state is added to $\mathcal{W}_{i+3}$ and therefore $\mathcal{W}_{i+3} = \mathcal{W}_{i+2}$ is a fixed point which is a contradiction. □

The proof for safety objectives is very similar, except that the states are removed from the sets $\mathcal{W}_0, \cdots, \mathcal{W}_d$ during the symbolic algorithm instead of being added. Since every time a player-2 state is added to (or removed from) the set of winning states, all the player-2 states with the same valuation over the variables $\mathcal{V}$ are added to (or removed from) the set of winning states, and there is only $|\Sigma_{\mathcal{V}}|$ player-2 states with different valuations over the variables $\mathcal{V}$, it follows that the symbolic algorithm for computing the set of winning states will terminate with at most $O(3|\Sigma_{\mathcal{V}}|) = O(|\Sigma_{\mathcal{V}}|)$ number of symbolic steps. □

Note that the upper-bound on the number of symbolic steps in Algorithm 4.2 is independent from the variables $\mathcal{V}^{\mathcal{C}} \backslash \mathcal{V}$. This is partly because transitions from player-2 states in the control game structure do not depend on the current valuation over variables $t_e, \gamma_{\mathcal{C}}$, and $\mathcal{P}$. Thus, if a player-2 state with the valuation $v \in \Sigma_{\mathcal{V}}$ over variables $\mathcal{V}$ is winning, then *all* player-2 states with the same valuation v over $\mathcal{V}$ are winning. Roughly speaking, the symbolic algorithm for computing the set of winning states manipulates the set of player-2 states based on their valuations over $\mathcal{V}$. Note that there are only $|\Sigma_{\mathcal{V}}|$ player-2 states with different valuations over $\mathcal{V}$ in $\mathcal{G}^{\mathcal{C}}$. Besides, any infinite run in $\mathcal{G}^{\mathcal{C}}$ starting from a player-2 state has a special form where every player-2 state is followed by two player-1 states and then a player-2 state is visited, and this pattern repeats infinitely. Due to this special form, with every three symbolic steps, either the symbolic algorithm terminates by reaching a fix point that characterize the set of winning states, or a new set of player-2 states with some valuation v over $\mathcal{V}$ are discovered to be winning (or losing) by the symbolic algorithm. Hence, the number of symbolic steps is bounded by $O(3|\Sigma_{\mathcal{V}}|) = O(|\Sigma_{\mathcal{V}}|)$.

Example 4.3. *Consider the setting in Example 4.2. Let* $\phi_{init} = (x = 0)$ *be a set of initial states, and* $\Phi = \square(x \neq 2) \wedge \lozenge(x = 1)$ *be the objective. A control strategy enforcing the objective* Φ *is shown in Figure 4.5 by solid edges at player-2 states. Initially, at* q_{11}*, the composer chooses controller* $\mathcal{C}_1$ *with parameter value* $p = 0$*. Once* $\mathcal{C}_{1_{\downarrow 0}}$ *reaches a final state, the control is returned to the composer, and based on the current valuation over* x*,* $x = 1$ *in this example, the next controller,* $\mathcal{C}_2$*, and the next parameter valuation,* $p = 1$*, are chosen by*

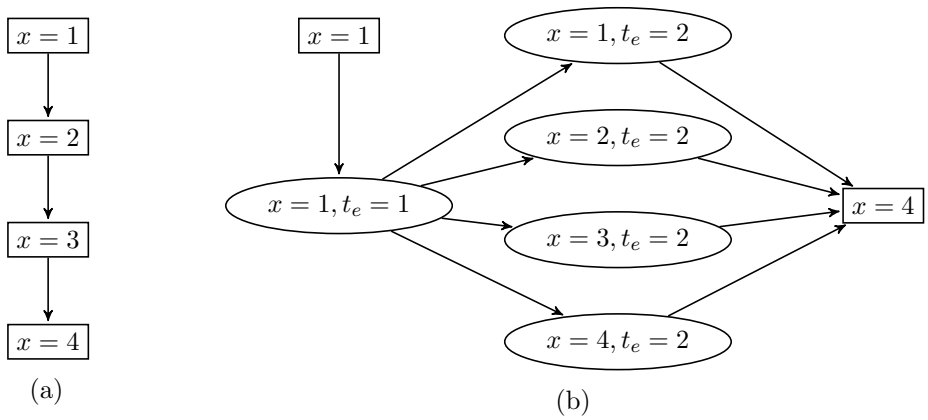


Figure 4.6: (a) Part of a run in a game structure where the controller takes the control at a state with $x = 1$ and increments x by 3. (b) Part of a control game structure capturing execution of the controller from a state with $x = 1$.

the composer. Intuitively, at states with $x = 0$, the composer increments x by selecting $\mathcal{C}_1$ and at states with $x = 1$, it decrements x by choosing $\mathcal{C}_2$.

Completeness. Note that Algorithm 4.2 is not complete as the interfaces provide an abstraction of the controllers and they might lack some information on the *sequence* of states that will be visited during the execution of the controller. As an example consider a game structure $\mathcal{G}$ with an integer variable x. Consider a controller $\mathcal{C}$ that starting from a state $x = p$ where p is a parameter, increments x by three, i.e., eventually $x = p + 3$. Assume that the controller does this by incrementing x one by one in three consecutive steps. Figure 4.6a shows a part of a run of $\mathcal{G}$ starting from a state where $x = 1$ and applying the controller $\mathcal{C}_{\downarrow p=1}$. For simplicity, we assume that all states in $\mathcal{G}$ are player-2 states (represented by boxes). The interface $\mathcal{I}_\mathcal{C}$ of the controller $\mathcal{C}$ is defined as $\mathcal{I}_\mathcal{C} = (\phi_{init_\mathcal{C}}, \phi_{inv_\mathcal{C}}, \phi_{f_\mathcal{C}})$ where $\phi_{init_\mathcal{C}} = (x = p)$, $\phi_{inv_\mathcal{C}} = \bigvee_{i=p}^{p+3} x = i$, and $\phi_{f_\mathcal{C}} = (x = p + 3)$. That is, starting from a parametric state $x = p$ and using the controller $\mathcal{C}$, any state $x \in [p..p + 3]$ can be visited, and eventually x is incremented by 3. Note that here we removed the constraints concerning the set of admissible parameter values from the interface to keep the example simple. Figure 4.6b shows part of a control game structure obtained from $\mathcal{I}_\mathcal{C}$ from any state with $x = 1$. To keep the figure simple, we removed the parameters and variables corresponding to the controllers, and similar to Example 4.2, player-2 states are grouped together based on their valuations over x. Let $\phi_{init} = (x = 1)$ and $\Phi = \Diamond(x = 3)$ specify the initial state and the objective of the system, respectively. It is easy to see that using controller $\mathcal{C}$ guarantees

visiting the state $x = 3$ on its path to the final state $x = 4$. However, there is no control strategy over the control game structure that guarantees visiting $x = 3$ as player-1 can avoid the state with $x = 3$ in the control game structure. Intuitively, the sequence of states $(x = 1)(x = 2)(x = 3)(x = 4)$ is "lost" in the control game structure. This loss of information is the cost paid for having a simpler control game structure.

For a given objective Φ, completeness of the framework can be achieved by analyzing the controllers and enriching the interfaces, or in the extreme case by having interfaces that exactly capture the possible outcomes of applying corresponding controllers. However, our main emphasis is on simplicity and separating the two design layers, the parametric controller synthesis and synthesis from a library of parametric controllers. Controllers in our framework can be viewed as *black-boxes* where their input-output behavior is provided through their simple interfaces. An alternative view is to see the controllers as *white-boxes* and extract more information from them. The trade-off is that the former approach is simpler and more computationally efficient as the control game structure is simpler and requires less number of symbolic steps, while the latter guarantees completeness.

4.5 Case Study

In this section we apply the methods developed in Sections 4.3 and 4.4 to an autonomous vehicle case study. Consider a network of one-way streets connected via intersections as shown in Figure 4.1. Assume there is a controlled autonomous vehicle V_1 initially positioned in the grid marked with s_0. Also assume there is an uncontrolled vehicle V_2 initially at the grid-cell s^*. Each vehicle has actions move-forward, back-up, turn-left, turn-right, and stop that moves it one step forward, backward, to the left, to the right, and leaves its position unchanged, respectively. The goal is to synthesize a controller for V_1 that can guide it from the initial position s to the final destination d while obeying the traffic laws (e.g., moving in the specified directions of streets, no stopping inside an intersection, etc.) and avoiding collision with V_2 and static obstacles. We assume that the uncontrolled vehicle respects traffic laws by always moving in the specified directions for the streets. We implement our algorithms symbolically in Java and using BDD package JDD [Vah]. We first specify and synthesize a set of parametric reactive controllers that guarantee advancing the vehicle in north, south, west, and east directions while avoiding collision with static obstacles and the other vehicle. We then synthesize a control strategy that instantiates and composes these

controllers to navigate V_1 safely from initial position to the destination.

Synthesis of Parametric Controllers. We denote the location of the vehicle V_i at any time-step with (x_i, y_i) for $i = 1, 2$. We specify four parametric controllers that can move the car in different directions: Controller $\mathcal{C}_1$ ($\mathcal{C}_2$) with specified controller interface $\mathcal{I}_1$ ($\mathcal{I}_2$) that moves the car three steps toward east (west, respectively), and controller $\mathcal{C}_3$ ($\mathcal{C}_4$) with specified controller interface $\mathcal{I}_3$ ($\mathcal{I}_4$) that advance the car one step toward north (south, respectively.) More specifically, let a, b be two parameters. Let $\phi_{init} = (x_1 = a \wedge y_1 = b)$ be the parametric initial state. Let $\phi_{inv} = (x_1 \neq x_2 \vee y_1 \neq y_2)$, i.e., no collision between vehicles. Finally, let $\phi_{f_1} = (x = a + 3, y = b)$, $\phi_{f_2} = (x = a - 3, y = b)$, $\phi_{f_3} = (x = a, y = b + 1)$, and $\phi_{f_4} = (x = a, y = b - 1)$ specify moving in different directions. Controllers are specified by interfaces $\mathcal{I}_{\mathcal{C}_i} = (\phi_{init}, \phi_{inv}, \phi_{f_i})$. Algorithm 4.1 is then used to synthesize parametric controllers, their corresponding interfaces and the set of admissible parameter values.

Synthesis of Control Strategy. Once a library of parametric reactive controllers $\Gamma_{\mathcal{C}}$ and their corresponding interfaces $\Gamma_{\mathcal{I}_{\mathcal{C}}}$ are formed, we use Algorithm 4.2 to synthesize a control strategy for the controlled vehicle. A synthesized control strategy instantiates and applies the controller $\mathcal{C}_1$ consecutively to advance the vehicle toward east and finally bring it to the position marked by s_6, from which controller $\mathcal{C}_3$ is instantiated and employed consecutively to take the vehicle to its destination. More specifically, at any position marked by s_i for $i = 0, \cdots, 5$, controller $\mathcal{C}_1$ is instantiated and becomes active, and it guarantees to eventually advance the controlled vehicle to the next position s_{i+1}. Similarly, at any position marked by s_i for $i = 6, \cdots, 9$, controller $\mathcal{C}_3$ is instantiated and becomes active, and it eventually navigates the controlled vehicle to the next position s_{i+1} where $s_{10} = d$ is the final destination. The control strategy sets the parameters a and b according to the current state of the vehicle, i.e., if the current position of V_1 is $(x_1 = i, y_1 = j)$, the control strategy instantiates the controllers $\mathcal{C}_1$ and $\mathcal{C}_3$ by parameter valuation $(a, b) = (i, j)$.

5

Compositional Synthesis of Reactive Controllers for Decoupled Multi-Agent Systems

One of the main challenges in automated synthesis of systems is the scalability problem. This issue becomes more evident for multi-agent systems, as adding each agent can often increase the size of the state space exponentially. The pioneering work by Pnueli et. al [PR89] showed that reactive synthesis from LTL specifications is intractable which prohibited the practitioners from utilizing synthesis algorithms. Distributed reactive synthesis [PR90] and multi-player games of incomplete information [PRA01] are undecidable in general. Despite these discouraging results, recent advances in this growing research area have enabled automatic synthesis of interesting real-world systems [BJP$^+$12], indicating the potential of the synthesis algorithms for solving realistic problems. The key insight is to consider more restricted yet practically useful subclasses of the general problem.

In this chapter, we consider a special class of multi-agent systems that are referred to as *decoupled* and are inspired by robot motion planning, decentralized control [KBB06, DM06], and swarm robotics [ŞGBT08, STZ$^+$12] literature. Intuitively, in a decoupled multi-agent system the transition relations (or dynamics) of the agents are decoupled, i.e., at any time-step, agents can make decisions on what action to take based on their own local state. For example, an autonomous vehicle can decide to slow down or speed up based on its position, velocity, etc. However, decoupled agents are coupled through objectives, i.e., an agent may need to cooperate with other agents or react to their actions to fulfill a given objective (e.g., it would not be a wise decision for an autonomous vehicle to speed up when the front vehicle pushes the break if collision avoidance is an objective.) In our framework, multi-agent systems consist of a set of controlled and uncontrolled agents. Controlled agents

may need to cooperate with each other and react to the actions of uncontrolled agents in order to fulfill their objectives. Besides, controlled agents may be imperfect in the sense that they can only partially observe their environment, for example due to the limitations in their sensors. The goal is to synthesize controllers for each controlled agent such that the objectives are enforced in the resulting system.

To solve the controller synthesis problem for multi-agent systems one can directly construct the model of the system by composing those of the agents, and solve the problem centrally for the given objectives. However, the centralized method lack flexibility, since any change in one of the components requires the repetition of the synthesis process for the whole system. Besides the resulting system might be exponentially larger than the individual parts, making this approach infeasible in practice. Compositional reactive synthesis aims to exploit the structure of the system by breaking the problem into smaller and more manageable pieces and solving them separately. Then solutions to subproblems are merged and analyzed to find a solution for the whole problem. The existing structure in multi-agent systems makes them a potential application area for compositional synthesis techniques.

To this end, we propose a compositional framework for decoupled multi-agent systems based on automatic decomposition of objectives and compositional reactive synthesis using maximally permissive strategies [FJR11]. We assume that the objective of the system is given in conjunctive form. We make an observation that in many cases each conjunct of the global objective only refers to a small subset of agents in the system. We take advantage of this structure to decompose the synthesis problem: for each conjunct of the global objective, we only consider the agents that are involved, and compute the maximally permissive strategies for those agents with respect to the considered conjunct. We then intersect the strategies to remove potential conflicts between them, and project back the constraints to subproblems, solve them again with updated constraints, and repeat this process until the strategies become fixed.

We implement the algorithms symbolically using BDDs and apply them to a robot motion planning case study where multiple robots are placed on a grid-world with static obstacles and other dynamic, uncontrolled and potentially adversarial robots. We consider different objectives such as collision avoidance, keeping a formation and bounded reachability. We show that by taking advantage of the structure of the system, the proposed compositional synthesis algorithm can significantly outperform the centralized synthesis approach, both

from time and memory perspective, and can solve problems where the centralized algorithm is infeasible. Furthermore, using compositional algorithms we managed to solve synthesis problems for systems with multiple agents, more complex objectives and for grid-worlds of sizes that are much larger than the cases considered in similar works. Our findings show the potential of symbolic and compositional reactive synthesis methods as planning algorithms in presence of dynamically changing and possibly adversarial environment.

5.1 Decoupled Multi-Agent Systems

In this section we describe how we model decoupled multi-agent systems and formally state the problem that is considered in this chapter. Typically game structures arise from description of open systems in a particular language [AHK02]. In our framework, we use *agents* to specify a system in a modular manner. An agent $\mathtt{a} = (\mathtt{type}, \mathcal{I}, \mathcal{O}, \Lambda, \tau, \mathcal{OBS}, \gamma)$ is a tuple where $\mathtt{type} \in \{\text{controlled}, \text{uncontrolled}\}$ indicates whether the agent can be controlled or not, $\mathcal{O}$ ($\mathcal{I}$) is a set of output (input) variables that the agent can (cannot, respectively) control by assigning values to them, Λ is a set of actions for the agent, and τ is a predicate over $\mathcal{I} \cup \mathcal{O} \cup \Lambda \cup \mathcal{O}'$ that specifies the possible transitions of the agent where $\mathcal{O}'$ is the primed copies of the variables $\mathcal{O}$, $\mathcal{OBS}$ is a set of observable variables, and $\gamma : \Sigma_{\mathcal{OBS}} \to 2^{\Sigma_{\mathcal{I}\cup\mathcal{O}}}$ is the observation function that maps agent's observations to its corresponding set of states. Intuitively, τ defines what actions an agent can choose at any state $s \in \Sigma_{\mathcal{I}} \times \Sigma_{\mathcal{O}}$ and what are the possible next valuations over agent's output variables for the chosen action. That is, $(i, o, \ell, o') \models \tau$ for $i \in \Sigma_{\mathcal{I}}$, $o \in \Sigma_{\mathcal{O}}$, $\ell \in \Lambda$, and $o' \in \Sigma_{\mathcal{O}'}$ means that at any state s of the system with $s_{|\mathcal{I}} = i$ and $s_{|\mathcal{O}} = o$, the agent can take action ℓ, and a state with component o' is a possible successor. A *perfect agent* is an agent with $\mathcal{OBS} = \mathcal{I} \cup \mathcal{O}$ and $\gamma(s) = \{s\}$ for all $s \in \Sigma_{\mathcal{I}} \times \Sigma_{\mathcal{O}}$. We omit $(\mathcal{OBS}, \gamma)$ in the description of perfect agents. An agent $\mathtt{a}$ is called *local* if and only if its transition relation τ is a predicate over $\mathcal{O} \cup \Lambda \cup \mathcal{O}'$, i.e., it does not depend on any uncontrolled variable $v \in \mathcal{I}$.

A multi-agent system $\mathcal{M} = \{\mathtt{a}_1, \mathtt{a}_2, \cdots, \mathtt{a}_n\}$ is defined as a set of agents $\mathtt{a}_i = (\mathtt{type}_i, \mathcal{I}_i, \mathcal{O}_i, \Lambda_i, \tau_i, \mathcal{OBS}_i, \gamma_i)$ for $1 \leq i \leq n$. Let $\mathcal{V} = \bigcup_{i=1}^{n} \mathcal{O}_i$ be the set of agents' output variables. We assume that the set of output variables of agents are pairwise disjoint, i.e., $\forall 1 \leq i \leq n.\ \mathcal{O}_i \cap \mathcal{O}_j = \emptyset$, and the set of input variables $\mathcal{I}_i$ for each agent $\mathtt{a}_i \in \mathcal{M}$ is a subset of variables controlled by other agents, i.e., $\mathcal{I}_i \subseteq \mathcal{V} \backslash \mathcal{O}_i$. We further make some simplifying assumptions. First, we assume that all uncontrolled agents are perfect, i.e.,

uncontrolled agent has perfect information about the state of the system at any time-step. Second, we assume that all controlled agents are *cooperative* while uncontrolled ones can play adversarially, i.e., the controlled agents cooperate with each other and make joint decisions to enforce the global objective. Finally, we assume that the observation variables for controlled agents are pairwise disjoint, i.e., $\forall 1 \leq i \leq n.\ \mathcal{OBS}_i \cap \mathcal{OBS}_j = \emptyset$, and that each controlled agent has perfect knowledge about other controlled agents' observations. That is, controlled agents share their observations with each other. Intuitively, it is as if the communication between controlled agents is instantaneous and error-free, i.e., they have perfect communication and tell each other what they observe. This assumption helps us preserve the two-player game setting and to stay in a decidable subclass of the more general problem of multi-player games with partial information. Note that multi-player games of incomplete information are undecidable in general [PRA01].

In this chapter we focus on a special setting where all agents are local. A multi-agent system $\mathcal{M} = \{\mathtt{a}_1, \mathtt{a}_2, \cdots, \mathtt{a}_n\}$ is *dynamically decoupled* (or decoupled in short) iff all agents $\mathtt{a} \in \mathcal{M}$ are local. Intuitively, agents in a decoupled multi-agent system can choose their action based on their own local state and regardless of the local states of other agents in the system. That is, the availability of actions for each agent in any state of the system is only a function of the agent's local state. Such setting arises in many applications, e.g., robot motion planning, where possible transitions of agents are independent from each other. For example, how a robot moves around a room is usually based on its own characteristics and motion primitives [SRK+14]. Note that this does not mean that the controlled agents are completely decoupled, as the objectives might concern different agents in the system, e.g., collision avoidance objective for a system consisting of multiple controlled robots, which requires cooperation between agents.

In our framework, the user describes the agents and specifies the objective as a conjunctive LTL formula. From description of the agents, a game structure is obtained that encodes how the state of the system evolves. Formally, given a decoupled multi-agent system $\mathcal{M} = \mathcal{M}^{\mathtt{u}} \biguplus \mathcal{M}^{\mathtt{c}}$ partitioned into a set $\mathcal{M}^{\mathtt{u}} = \{\mathtt{u}_1, \cdots, \mathtt{u}_m\}$ of uncontrolled agents and a set $\mathcal{M}^{\mathtt{c}} = \{\mathtt{c}_1, \cdots, \mathtt{c}_n\}$ of controlled agents, the turn-based game structure $\mathcal{G}^{\mathcal{M}}$ induced by $\mathcal{M}$ is defined as $\mathcal{G}^{\mathcal{M}} = (\mathcal{V}, \Lambda, \tau, \mathcal{OBS}, \gamma)$ where $\mathcal{V} = \{t\} \cup \bigcup_{\mathtt{a} \in \mathcal{M}} \mathcal{O}_{\mathtt{a}}$ is the set of all variables in $\mathcal{M}$ with t as a turn variable, $\Lambda = \bigcup_{\mathtt{a} \in \mathcal{M}} \Lambda_{\mathtt{a}}$ is the set of actions, $\mathcal{OBS} = \bigcup_{\mathtt{c} \in \mathcal{M}^{\mathtt{c}}} \mathcal{OBS}_{\mathtt{c}}$ is the set of all observation variables of controlled agents (note that we assume all uncontrolled

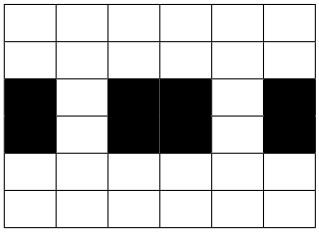


Figure 5.1: Grid-world with static obstacles

agents are perfect,) and τ and γ are defined as follows:

$$\tau = \tau_e \vee \tau_s,$$
$$\tau_e = t = 1 \wedge t' = 2 \wedge \bigwedge_{\mathtt{u} \in \mathcal{M}^{\mathtt{u}}} \tau_{\mathtt{u}} \wedge \bigwedge_{\mathtt{c} \in \mathcal{M}^{\mathtt{c}}} Same(\mathcal{O}_{\mathtt{c}}, \mathcal{O}'_{\mathtt{c}}),$$
$$\tau_s = t = 2 \wedge t' = 1 \wedge \bigwedge_{\mathtt{c} \in \mathcal{M}^{\mathtt{c}}} \tau_{\mathtt{c}} \wedge \bigwedge_{\mathtt{u} \in \mathcal{M}^{\mathtt{u}}} Same(\mathcal{O}_{\mathtt{u}}, \mathcal{O}'_{\mathtt{u}}), \text{ and}$$
$$\gamma = \bigwedge_{\mathtt{c} \in \mathcal{M}^{\mathtt{c}}} \gamma_{\mathtt{c}}$$

Intuitively, at each step, uncontrolled agents take actions consistent with their transition relations, and their variables get updated while the controlled agents' variables stay unchanged. Then the controlled agents react concurrently and simultaneously by taking actions according to their transition relations, and their corresponding variables get updated while the uncontrolled agents' variables stay unchanged.

Example 5.1. *Let R_1 and R_2 be two robots in an $n \times n$ grid-world similar to the one shown in Figure 5.1. Assume R_1 is an uncontrolled robot, whereas R_2 can be controlled. In the sequel, let i range over $\{1, 2\}$. At each time any robot R_i can move to one of its neighboring cells by taking an action from the set $\Lambda_i = \{up_i, down_i, right_i, left_i\}$. Furthermore, assume that R_2 has imperfect sensors and can only observe R_1 when R_1 is in one of its adjacent cells. Let (x_i, y_i) represent the position of robot R_i in the grid-world at any time*[5]*. We define $\mathcal{O}_i = \{x_i, y_i\}$ and $\mathcal{I}_i = \mathcal{O}_{3-i}$ as the output and input variables, respectively. Note that the controlled variables by one agent are the input variables of the other agent. Transition relation $\tau_i = \bigwedge_{\ell \in \Lambda_i} \tau_\ell$ is defined as conjunction of four parts corresponding to robot's action*

[5]Note that variables x_i and y_i are defined over a bounded domain and can be encoded by a set of Boolean variables. To keep the example simple, we use their bounded integer representation here.

where

$$\tau_{up_i} = (y_i > 1) \wedge up_i \wedge (y_i' \leftrightarrow y_i - 1) \wedge Same(x_i, x_i')$$
$$\tau_{down_i} = (y_i < n) \wedge down_i \wedge (y_i' \leftrightarrow y_i + 1) \wedge Same(x_i, x_i')$$
$$\tau_{left_i} = (x_i > 1) \wedge left_i \wedge (x_i' \leftrightarrow x_i - 1) \wedge Same(y_i, y_i')$$
$$\tau_{right_i} = (x_i < n) \wedge right_i \wedge (x_i' \leftrightarrow x_i - 1) \wedge Same(y_i, y_i')$$

Intuitively, each τ_ℓ for $\ell \in \Lambda_i$ specifies whether the action is available in the current state and what is its possible successors. For example, τ_{up_i} indicates that if R_i is not at the top row ($y_i > 1$), then the action up_i is available and if applied, in the next state the value of y_i is decremented by one and the value of x_i does not change. Next we define the observation function γ_2 for R_2. It is easier and more intuitive to define γ_2^{-1}, and since observations partition the state space $\gamma_2 = (\gamma_2^{-1})^{-1}$ is defined. Formally,

$$\gamma_2^{-1}(a,b,c,d) = \begin{cases} (a,b,c,d) & \text{if } a-1 \le c \le a+1 \wedge b-1 \le d \le b+1 \\ (\bot,\bot,c,d) & \text{otherwise} \end{cases}$$

Let $\mathcal{OBS}_2 = \{x_1^o, y_1^o, x_2^o, y_2^o\}$ where $x_1^o, y_1^o \in \{\bot, 1, 2, \cdots, n\}$ and $x_2^o, y_2^o \in \{1, \cdots, n\}$. Intuitively, R_2 observes its own local state perfectly. Furthermore, if R_1 is in one of its adjacent cells, its position is observed perfectly, otherwise, R_1 is away and its location cannot be observed. γ_2 can be symbolically encoded as $\bigvee_{o \in \Sigma_{\mathcal{OBS}}} (o \wedge \phi_{\gamma(o)})$ where $\phi_{\gamma(o)}$ is the predicate specifying the set $\gamma(o)$. Finally, we let $R_1 = (uncontrolled, \mathcal{I}_1, \mathcal{O}_1, \Lambda_1, \tau_1)$ and $R_2 = (controlled, \mathcal{I}_2, \mathcal{O}_2, \Lambda_2, \mathcal{OBS}_2, \gamma_2)$. Note that R_1 (R_2) is modeled as a perfect (imperfect, respectively) local agent.

The game structure $\mathcal{G}^{\mathcal{M}}$ of imperfect information corresponding to multi-agent system $\mathcal{M} = \{R_1, R_2\}$ is a tuple $\mathcal{G}^{\mathcal{M}} = (\mathcal{V}, \Lambda, \tau, \mathcal{OBS}, \gamma)$ where $\mathcal{V} = \{t\} \cup \mathcal{O}_1 \cup \mathcal{O}_2$, $\Lambda = \Lambda_1 \cup \Lambda_2$, $\tau = \tau_e \vee \tau_s$, $\tau_e = t = 1 \wedge t' = 2 \wedge \tau_1 \wedge Same(\mathcal{O}_2, \mathcal{O}_2')$, $\tau_s = t = 2 \wedge t' = 1 \wedge \tau_2 \wedge Same(\mathcal{O}_1, \mathcal{O}_1')$, $\mathcal{OBS} = \mathcal{OBS}_2$, and $\gamma = \gamma_2$. □

We now formally define the problem we consider in this chapter.

Problem 5.1. *Given a decoupled multi-agent system $\mathcal{M} = \mathcal{M}^{\mathtt{u}} \biguplus \mathcal{M}^{\mathtt{c}}$ partitioned into uncontrolled $\mathcal{M}^{\mathtt{u}} = \{\mathtt{u}_1, \cdots, \mathtt{u}_m\}$ and controlled agents $\mathcal{M}^{\mathtt{c}} = \{\mathtt{c}_1, \cdots, \mathtt{c}_n\}$, a predicate ϕ_{init} specifying an initial state, and an objective $\Phi = \Phi_1 \wedge \cdots \wedge \Phi_k$ as conjunction of*

$k \geq 1$ LTL formulas Φ_i, compute strategies $\mathtt{S}_1, \cdots, \mathtt{S}_n$ for controlled agents such that the strategy $\mathtt{S} = \mathtt{S}_1 \otimes \cdots \otimes \mathtt{S}_n$ defined as composition of the strategies is winning for the game $(\mathcal{G}^{\mathcal{M}}, \phi_{init}, \Phi)$, where $\mathcal{G}^{\mathcal{M}}$ is the game structure induced by $\mathcal{M}$.

5.2 Compositional Controller Synthesis

We now explain our solution approach for Problem 5.1 stated in Section 5.1. Algorithm 5.1 summarizes the steps for compositional synthesis of strategies for controlled agents in a multi-agent system. It has three main parts. First the synthesis problem is automatically decomposed into subproblems by taking advantage of the structure in the multi-agent system and given objective. Then the subproblems are solved separately and their solutions are composed. Composition may restrict the possible actions that are available for agents at some states. The composition is then projected back to each subproblem and the subproblems are solved again with new restrictions. This process is repeated until either a subgame becomes unrealizable, or computed solutions for subproblems reach a fixed point. Finally, a set of strategies, one for each controlled agent, is extracted by decomposing the strategy obtained in the previous step. Next, we explain Algorithm 5.1 in more detail.

5.2.1 Decomposition of the Synthesis Problem

The synthesis problem is decomposed into subproblems in lines $2-9$ of Algorithm 5.1. The main idea behind this decomposition is that, in many cases, each conjunct Φ_i of the objective Φ only refers to a small subset of agents. This observation is utilized to obtain a game structure from description of those agents that are *involved* in Φ_i, i.e., only agents are considered to form and solve a game with respect to Φ_i that are relevant. In other words, each subproblem corresponds to a conjunct Φ_i of the global objective Φ and the game structure obtained from agents involved in Φ_i.

For each conjunct Φ_i, $1 \leq i \leq k$, Algorithm 5.1 first obtains the set $\mathcal{INV}_i$ of involved agents using the procedure **Involved**. Formally, let $\mathcal{V}_{\Phi_i} \subseteq \mathcal{V}$ be the set of variables appearing in Φ_i's formula. The set of involved agents are those agents whose controlled variables appear in the conjunt's formula, i.e., $\textbf{Involved}(\Phi_i) = \{a \in \mathcal{M} \mid \mathcal{O}_a \cap \mathcal{V}_{\Phi_i} \neq \emptyset\}$.

A game structure $\mathcal{G}_i$ is then obtained from the description of the agents $\mathcal{INV}_i$ using the procedure **CreateGameStructure** as explained in Section 5.1. The projection ϕ^i_{init} of the predicate ϕ_{init} with respect to the involved agents is computed next. The procedure **Project**

Algorithm 5.1: Compositional Controller Synthesis

Input: A decoupled multi-agent system $\mathcal{M} = \{\mathtt{u}_1, \cdots, \mathtt{u}_m, \mathtt{c}_1, \cdots, \mathtt{c}_n\}$, ϕ_{init} specifying initial state, and an objective $\Phi = \Phi_1 \wedge \cdots \wedge \Phi_k$

Output: A set of strategies $(\mathcal{S}_1, \cdots, \mathcal{S}_n)$ one for each controlled agent, if one exists

1 /* Decompose the problem based on agents' involvement in conjuncts*/
2 **for** *all* $\Phi_i, 1 \leq i \leq k$ **do**
3 $\quad \mathcal{INV}_i :=$ **Involved**(Φ_i);
4 $\quad \mathcal{G}_i :=$ **CreateGameStructure**$(\mathcal{INV}_i)$;
5 $\quad \mathcal{X}_i := \bigcup_{a \in \mathcal{INV}_i} \mathcal{O}_a$; /* the set of variables controlled by involved agents */
6 $\quad \phi^i_{init} :=$ **Project**$(\phi_{init}, \mathcal{X}_i)$;
7 $\quad \mathcal{G}^K_i :=$ **CreateKnowledgeGameStructure**$(\mathcal{G}_i)$;
8 $\quad (\mathcal{G}^d_i, \phi^i_{init}) :=$ **ToSafetyGame**$(\mathcal{G}^K_i, \phi^i_{init}, \Phi_i)$;
9 /*Compositional synthesis*/
10 **while** `true` **do**
11 $\quad$ **for** $i = 1 \cdots k$ **do**
12 $\quad\quad \mathcal{S}^d_i :=$ **SolveSafetyGame**$(\mathcal{G}^d_i, \phi^i_{init})$;
13 $\quad \mathcal{S} := \bigotimes_{i=1}^{m} \mathcal{S}^d_i$; /* compose the strategies */
14 $\quad$ **for** $i = 1 \cdots k$ **do**
15 $\quad\quad$ Let $\mathcal{Y}_i = \mathcal{V}^d_i \cup \Lambda^d_i$ be the set of variables and actions in $\mathcal{G}^d_i$;
16 $\quad\quad \mathcal{C}_i :=$ **Project**$(\mathcal{S}, \mathcal{Y}_i)$; /* project the strategies */
17 $\quad$ **if** $\forall 1 \leq i \leq k, \mathcal{S}^d_i = \mathcal{C}_i$ **then**
18 $\quad\quad$ break; /* a fixed point is reached over strategies */
19 $\quad$ **for** $i = 1 \cdots k$ **do**
20 $\quad\quad \mathcal{G}^d_i := \mathcal{G}^d_i[\mathcal{C}_i]$; /* Restrict the subgames for the next iteration */
21 $(\mathtt{S}_1, \cdots, \mathtt{S}_n) :=$**Extract**$(\mathcal{S})$;
22 return $(\mathtt{S}_1, \cdots, \mathtt{S}_n)$;

takes a predicate ϕ over variables $\mathcal{V}_\phi$ and a subset $\mathcal{X} \subseteq \mathcal{V}_\phi$ of variables as input, and projects the predicate with respect to the given subset. Formally, $\textbf{Project}(\phi, \mathcal{X}) = \left\{ s_{|\mathcal{X}} \mid s \in \Sigma_{\mathcal{V}_\phi} \right\}$.

The knowledge game structure $\mathcal{G}^K_i$ corresponding to $\mathcal{G}_i$ is obtained at line 7. Note that this step is not required if the system only includes agents that can observe the state of the game perfectly at any time-step. Finally, the objective Φ_i is transformed into a game structure using the algorithms in [FJR11, Ehl12] and composed with $\mathcal{G}^K_i$ to obtain a safety game $(\mathcal{G}^d_i, \phi^i_{init})$. The result of decomposition phase is k safety games $\left\{(\mathcal{G}^d_1, \phi^1_{init}), \cdots, (\mathcal{G}^d_k, \phi^k_{init})\right\}$ that form the subproblems for the compositional synthesis phase.

Example 5.2. *Let R_i for $i = 1, \cdots, 4$ be four robots in an $n \times n$ grid-world, where R_4 is uncontrolled and other robots are controlled. For simplicity, assume that all agents are*

perfect. At each time-step any robot R_i can move to one of its neighboring cells by taking an action from the set $\{up_i, down_i, right_i, left_i\}$ with their obvious meanings. Consider the following objective $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Phi_{12} \wedge \Phi_{23}$ where Φ_i for $i = 1, 2, 3$ specifies that R_i must not collide with R_4, and Φ_{12} (Φ_{23}) specifies that R_1 and R_2 (R_2 and R_3, respectively) must avoid collision with each other. Sub-formulas Φ_i, $i = 1, 2, 3$, only involve agents R_i and R_4, i.e., $\mathcal{INV}(\Phi_i) = \{R_i, R_4\}$. Therefore, the game structures $\mathcal{G}_i$ induced by agents R_i and R_4 are composed with the game structure computed for Φ_i to form a subproblem as a safety game. Similarly, we obtain safety games for objectives Φ_{12} and Φ_{23} with $\mathcal{INV}(\Phi_{12}) = \{R_1, R_2\}$ and $\mathcal{INV}(\Phi_{23}) = \{R_2, R_3\}$, respectively. □

Remark 5.1. *The decomposition method used here is neither the only way to decompose the problem, nor necessarily optimal. More efficient decomposition techniques can be used to obtain quicker convergence in Algorithm 5.1 for example by different grouping of conjuncts. Nevertheless, the decomposition technique explained above is simple and proved effective in our experiments.*

5.2.2 Compositional Synthesis

The safety games obtained in the decomposition phase are compositionally solved in lines $9 - 21$ of Algorithm 5.1. At each iteration of the main loop, the subproblems $(\mathcal{G}_i^d, \phi_{init}^i)$ are solved, and a maximally permissive strategy $\mathcal{S}_i^d$ is computed for them, if one exists. Computed strategies are then composed in line 11 of Algorithm 5.1 to obtain a strategy $\mathcal{S}$ for the whole system. The strategy $\mathcal{S}$ is then projected back to sub-games, and it is checked whether all the projected strategies are equivalent to the strategies computed for the subproblems. If that is the case, the main loop terminates and $\mathcal{S}$ is winning for the game $(\mathcal{G}^d, \phi_{init})$ where $(\mathcal{G}^d, \phi_{init})$ is the safety game associated with the multi-agent system $\mathcal{M}$ and objective Φ. Otherwise, at least one of the subproblems needs to be restricted. Each sub-game is restricted by the computed projection, and the process is repeated. The loop terminates either if a subproblem becomes unrealizable at some iteration, or if permissive strategies $\mathcal{S}_1, \cdots, \mathcal{S}_k$ reach a fixed point. In the latter case, a set of strategies, one for each controlled agent is extracted from $\mathcal{S}$ as explained below.

5.2.3 Computing Strategies for the Agents

Let $\mathcal{V}^{\otimes} = \bigcup_{i=1}^{k} \mathcal{V}_{\mathcal{G}_i^d}$ be the set of all variables used to encode the game structures $\mathcal{G}_i^d$, and $\Lambda^c = \Lambda_{\mathtt{c}_1} \times \cdots \times \Lambda_{\mathtt{c}_n}$ be the set of actions of the controlled agents. Once a permissive strategy $\mathcal{S} : \Sigma_{\mathcal{V}^{\otimes}} \to 2^{\Lambda^c}$ is computed, a winning strategy $\mathtt{S}_d : \Sigma_{\mathcal{V}^{\otimes}} \to \Lambda^c$ is obtained from $\mathcal{S}$ by restricting the non-deterministic action choices of the controlled agents to a single action. The strategy $\mathtt{S}_d$ is then decomposed into strategies $\mathtt{S}_1 : \Sigma_{\mathcal{V}^{\otimes}} \to \Lambda_{\mathtt{c}_1}, \cdots, \mathtt{S}_n : \Sigma_{\mathcal{V}^{\otimes}} \to \Lambda_{\mathtt{c}_n}$ for the agents simply by projecting the actions over system transitions to their corresponding agents. Formally, for any $s \in \Sigma_{\mathcal{V}^{\otimes}}$ such that $\mathcal{S}(s)$ is defined, let $\mathtt{S}_d(s) = \sigma \in \mathcal{S}(s)$ where $\sigma = (\sigma_1, \cdots, \sigma_n) \in \Lambda^c$ is an arbitrary action chosen from possible actions permitted by $\mathcal{S}$ in the state s. Agents' strategies are defined as $\mathtt{S}_i(s) = \sigma_i$ for $i = 1, \cdots, n$. Note that we assume each controlled agent has perfect knowledge about other controlled agents' observations. The following theorem establishes the correctness of Algorithm 5.1.

Theorem 5.1. *Algorithm 5.1 is sound.*

Proof. Note that Algorithm 5.1 always terminates, that is because either eventually a fixed point over strategies is reached, or a sub-game becomes unrealizable which indicates that the objective cannot be enforced. Consider the permissive strategies $\mathcal{S}_i^d$ and their projections $\mathcal{C}_i$. We have $\mathcal{C}_i(s) \subseteq \mathcal{S}_i^d(s)$ for any $s \in \Sigma_{\mathcal{V}}$, and as a result of composing and projecting intermediate strategies, we will obtain more restricted sub-games. As the state space and available actions in any state is finite, at some point, either a sub-game becomes unrealizable because the system player becomes too restricted and cannot win the game, or all strategies reach a fixed point. Therefore, the algorithm always terminates.

We now show that Algorithm 5.1 is sound, i.e., if it computes strategies $(\mathtt{S}_1, \cdots, \mathtt{S}_n)$, then the strategy $\mathtt{S} = \bigotimes_{i=1}^{n} \mathtt{S}_i$ is a winning strategy in the game $(\mathcal{G}^{\mathcal{M}}, \phi_{init}, \Phi)$, where $\mathcal{G}^{\mathcal{M}}$ is the game structure induced by $\mathcal{M}$. Let $\mathcal{S}^* = \bigotimes_{i=1}^{k} \mathcal{S}_i^d$ be the fixed point reached over the strategies. First note that any run in $\mathcal{G}_i^d[\mathcal{S}_i^d]$ starting from a state $s \models \phi_{init}^i$ for $1 \leq i \leq k$ satisfies the conjunct Φ_i since $\mathcal{S}_i^d$ is winning in the corresponding safety game. That is, the restriction of the game structure $\mathcal{G}_i^d$ to the strategy $\mathcal{S}_i^d$ satisfies Φ_i. Consider any run $\pi = s_0 s_1 s_2 \cdots$ in the restricted game structure $\mathcal{G}^d[\mathcal{S}^*]$ starting from the initial state $s_0 \models \phi_{init}$ where $\mathcal{G}^d = \bigotimes_{i=1}^{k} \mathcal{G}_i^d$. Let $\pi^i = s_0^i s_1^i s_2^i \cdots$ for $1 \leq i \leq k$ be the projection of π with respect to variables $\mathcal{V}_i^d$ of the game structure $\mathcal{G}_i^d$, i.e., $s_j^i = s_{j_{|\mathcal{V}_i^d}}$ for $j \geq 0$. Since $s_0^i \models \phi_{init}^i$ and $\mathcal{S}_i^d$ is equivalent to the projection of $\mathcal{S}^*$ with respect to variables and actions

in the game structure $\mathcal{G}_i^d$, it follows that π^i is a winning run in the safety game $(\mathcal{G}_i^d[\mathcal{S}_i^d], \Phi_i)$, i.e., $\pi^i \models \Phi_i$. As $\pi^i \models \Phi_i$ for $1 \leq i \leq k$, we have $\pi \models \Phi = \bigwedge_{i=1}^{k} \Phi_i$. It follows that $\mathcal{S}^*$ is winning in the safety game $(\mathcal{G}^d, \phi_{init})$. Moreover, $\mathcal{S}^*$ is also winning with respect to the original game as $(\mathcal{G}^d, \phi_{init})$ is the safety game associated with $(\mathcal{G}^{\mathcal{M}}, \phi_{init}, \Phi)$ [FJR11]. It is easy to see that the set $(\mathtt{S}_1, \cdots, \mathtt{S}_n)$ of strategies extracted from $\mathcal{S}^*$ by Algorithm 5.1 is winning for the game $(\mathcal{G}^{\mathcal{M}}, \phi_{init}, \Phi)$. □

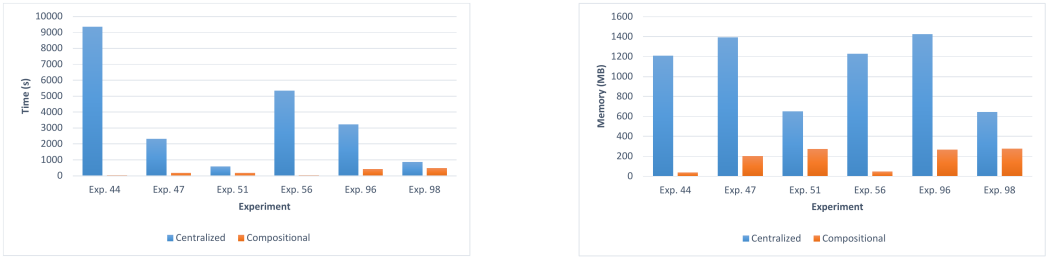
Remark 5.2. *In [FJR11] it is shown that bounded synthesis is complete by proving the existence of a sufficiently large bound. Following their result, it can be shown that Algorithm 5.1 is also complete. However, in practice, the required bound is rather high and instead an incremental approach is used for synthesis.*

Remark 5.3. *Algorithm 5.1 is different from another compositional algorithm proposed in [FJR11] in two ways. First, it composes maximally permissive strategies in contrast to composing game structures as proposed in [FJR11]. The advantage is that strategies usually have more compact symbolic representations compared to game structures.*[6] *Second, in the compositional algorithm in [FJR11], sub-games are composed and a symbolic step, i.e., a post- or pre-image computation, is performed over the composite game. In our experiments, performing a symbolic step over composite game resulted in a poor performance, often worse than the centralized algorithm. Algorithm 5.1 removes this bottleneck as it is not required in our setting. This leads to a significant improvement in performance since image and pre-image computations are typically the most expensive operations performed by symbolic algorithms [BGS06].*

5.3 Case Study

We now demonstrate the techniques on a robot motion planning case study similar to those that can be found in the related literature (e.g., [KGFP09, SRK+14, AKK11]). Consider a square grid-world with some static obstacles similar to the one depicted in Figure 5.1. We consider a multi-agent system $\mathcal{M} = \{\mathtt{u}_1, \cdots, \mathtt{u}_m, \mathtt{c}_1, \cdots, \mathtt{c}_n\}$ with uncontrolled robots $\mathcal{M}^u = \{\mathtt{u}_1, \cdots, \mathtt{u}_m\}$ and controlled ones $\mathcal{M}^c = \{\mathtt{c}_1, \cdots, \mathtt{c}_n\}$. At any time-step, any controlled robot $\mathtt{c}_i$ for $1 \leq i \leq n$ can move to one of its neighboring cells by taking actions $up_i, down_i, left_i,$ and $right_i$, or it can stay put by taking the action *stop*. Any uncontrolled

[6] Strategies are mappings from states to actions while game structures include more variables and typically have more complex BDD representation as they refer to states, actions, and next states.



(a) Time

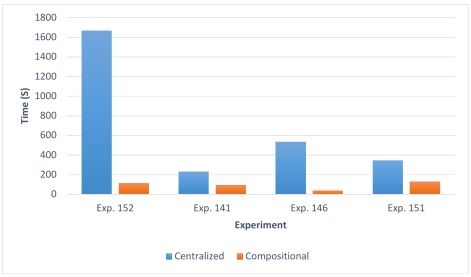
(b) Memory

Figure 5.2: Comparison of centralized and compositional approaches on a robot motion planning case study with perfect agents

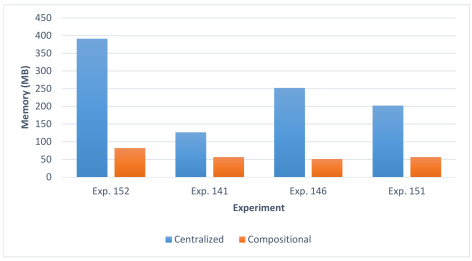
robot $\mathtt{u}_j$ for $1 \leq j \leq m$ stays on the same row where they are initially positioned, and at any time-step can move to their left or right neighboring cells by taking actions $left_j$ and $right_j$, respectively. We consider the following objectives for the systems, (Φ_1) collision avoidance, i.e., controlled robots must avoid collision with static obstacles and other robots, (Φ_2) formation maintenance, i.e., each controlled robot $\mathtt{c}_i$ must keep a linear formation (same horizontal or vertical coordinate) at all times with the subsequent controlled robot $\mathtt{c}_{i+1}$ for $1 \leq i < n$, (Φ_3) bounded reachability, i.e., controlled robots must reach the bottom row in a pre-specified number of steps. We consider two settings. First we assume all agents are perfect, i.e., all agents have full-knowledge of the state of the system at any time-step. Then we assume controlled agents are imperfect and can observe uncontrolled robots only if they are nearby and occupying an adjacent cell, similar to Example 5.1.

We apply two different methods to synthesize strategies for the agents. In the *Centralized* method, a game structure for the whole system is obtained first, and then a winning strategy is computed with respect to the considered objective. In the *Compositional* approach, the strategy is computed compositionally using Algorithm 5.1. We implemented the algorithms in Java using the BDD package JDD [Vah]. The experiments are performed on an Intel core i7 3.40 GHz machine with 16GB memory. In our experiments, we vary the number of uncontrolled and controlled agents, size of the grid-world, and the objective of the system as shown in Tables 5.1 and 5.2. The columns show the number of uncontrolled and controlled robots, considered objective, size of the grid-world, number of variables in the system, and the time and memory usage for different approaches, respectively. Furthermore, we define $\Phi_{12} = \Phi_1 \wedge \Phi_2$, $\Phi_{13} = \Phi_1 \wedge \Phi_3$, and $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$.

Multi-agent systems with perfect agents. Table 5.1 shows some of our experimental



(a) Time



(b) Memory

Figure 5.3: Comparison of centralized and compositional approaches on a robot motion planning case study with imperfect agents

Table 5.1: Experimental results for systems with perfect agents

					Centralized		Compositional	
$\lvert\mathcal{M}^u\rvert$	$\lvert\mathcal{M}^c\rvert$	Obj.	size	$\lvert\mathcal{V}\rvert$	time	mem (MB)	time	mem (MB)
1	1	Φ_1	64×64	52	**72 ms**	6.6	105 *ms*	6.6
1	1	Φ_1	128×128	60	**93 ms**	6.6	101 *ms*	6.6
1	2	Φ_{13}	16×16	79	14.9 *min*	365.5	**4.2 s**	**19.3**
1	2	Φ_{13}	32×32	95	mem out	mem out	**34.4 s**	**50.8**
1	2	Φ	16×16	79	400.3 *s*	239.7	**5.1 s**	**19.4**
1	2	Φ	32×32	95	155.8 *min*	1209	**33.1 s**	**38.3**
1	3	Φ_{13}	4×4	66	22 *s*	50.8	**0.8 s**	**6.8**
1	3	Φ_{13}	8×8	88	mem out	mem out	**98.4 s**	**101.2**
2	1	Φ	8×8	51	106.4 *s*	322	**33 ms**	**6.6**
2	1	Φ	128×128	107	mem out	mem out	**3.5s**	**6.7**
2	2	Φ	4×4	56	3.2 *s*	19.4	**201 ms**	**6.6**
2	2	Φ	8×8	76	10.6 *min*	460	**14.4 s**	**19.4**
2	3	Φ_{13}	4×4	75	19.1 *min*	497.8	**8.4 s**	**25.9**
2	3	Φ_{13}	8×8	101	mem out	mem out	**30.2 min**	**800.2**
2	3	Φ	8×8	101	mem out	mem out	**12.7 min**	**302.6**

results for the setting where all agents are perfect. Note that the compositional algorithm does not always perform better than the centralized alternative. Indeed, if the conjuncts of objectives involve a large subset of agents, compositional algorithm comes closer to the centralized algorithm. Intuitively, if the agents are "strongly" coupled, the overhead introduced by compositional algorithm is not helpful, and the centralized algorithm performs better. For example, when the system consists of a controlled robot and an uncontrolled one along with a single safety objective, compositional algorithm coincides with the centralized one, and centralized algorithm performs slightly better. However, if the subproblems are "loosely" coupled, which is the case in many practical problems, the compositional algorithm significantly outperforms the centralized one, both from time and memory perspective, as

Table 5.2: Experimental results for systems with imperfect agents

					Centralized		Compositional	
$\|\mathcal{M}^u\|$	$\|\mathcal{M}^c\|$	Obj.	size	$\|\mathcal{V}\|$	time	mem (MB)	time	mem (MB)
1	2	Φ_{12}	4×4	127	1.7 s	6.7	**0.6 s**	6.7
1	2	Φ_{12}	6×6	235	28.6 s	31.9	**10.2 s**	**19.3**
1	2	Φ_{12}	8×8	235	229.7 s	126.6	**95 s**	**57.1**
1	2	Φ_{12}	9×9	375	time out	time out	**306 s**	**94.9**
1	2	Φ_{12}	10×10	375	time out	time out	**9.7 min**	**176.7**
1	2	Φ_{13}	4×4	143	1.4 s	6.7	**303 ms**	6.7
1	2	Φ_{13}	6×6	255	38.2 s	57.1	**5 s**	**13**
1	2	Φ_{13}	8×8	255	8.9 min	252.2	**38.3 s**	**51**
1	2	Φ_{13}	9×9	395	time out	time out	**114.9 s**	**88.6**
1	2	Φ_{13}	10×10	395	time out	time out	**279.9 s**	**157.8**
1	2	Φ	4×4	143	2.3 s	6.7	**0.7 s**	**6.7**
1	2	Φ	6×6	255	46.2 s	50.8	**10 s**	**19.3**
1	2	Φ	8×8	255	344.5 s	202.1	**129.9 s**	**57.1**
1	2	Φ	9×9	395	time out	time out	**309.9 s**	**101.2**
1	2	Φ	10×10	395	time out	time out	**9.6 min**	**176.7**
1	3	Φ_1	4×4	186	144.3 s	69.7	**0.9 s**	**6.7**
1	3	Φ_1	6×6	346	time out	time out	**17.7 s**	**38.2**
1	3	Φ_1	8×8	346	time out	time out	**190.9 s**	**176.7**
1	3	Φ_1	10×10	554	time out	time out	**24.6 min**	**730.6**
1	3	Φ_{13}	4×4	210	265.8 s	214.5	**0.9 s**	**6.7**
1	3	Φ_{13}	6×6	376	time out	time out	**49.2 s**	**57.1**
1	3	Φ_{13}	8×8	376	time out	time out	**483.9 s**	**214.5**
1	3	Φ_{13}	9×9	584	time out	time out	**31.7 min**	**441.1**
1	3	Φ	6×6	376	time out	time out	**36 s**	**50.8**
1	3	Φ	8×8	376	time out	time out	**343.4 s**	**201.9**
1	3	Φ	10×10	584	time out	time out	**39.6 min**	**774.7**

we increase the number of agents and make the objectives more complex, and it can solve problems where the centralized algorithm is infeasible. Figure 5.2 shows the computation time and memory usage in some of our experiments where both centralized and compositional algorithms successfully computed strategies. Tables 5.3 and 5.4 show a more detailed version of our experimental results for multi-agent systems with perfect agents.

Multi-agent systems with imperfect controlled agents. Not surprisingly, scalability is a bigger issue when it comes to games with imperfect information due to the subset construction procedure, which leads to yet another reason for compositional algorithm to perform better than the centralized alternative. Table 5.2 shows some of our experimental results for the setting where controlled agents are imperfect. While the centralized approach fails to compute the knowledge game structure due to the state explosion problem, the compositional algorithm performs significantly better by decomposing the problem and per-

forming subset construction on smaller and more manageable game structures of imperfect information. Figure 5.3 shows the computation time and memory usage in some of our experiments where both centralized and compositional algorithms successfully computed strategies. Tables 5.5 and 5.6 show a more detailed version of our experimental results for multi-agent systems with imperfect agents.

Table 5.3: Evaluation of approaches on a robot motion planning case study with perfect agents

						Centralized		Compositional	
Ex. #	$\lvert\mathcal{M}^u\rvert$	$\lvert\mathcal{M}^e\rvert$	objective	size	$\lvert\mathcal{V}\rvert$	time	mem (MB)	time	mem (MB)
1	1	1	Φ_1	4×4	20	36 ms	0.3	44 ms	0.3
2	1	1	Φ_1	8×8	28	6 ms	0.3	20 ms	0.3
3	1	1	Φ_1	16×16	36	24 ms	0.3	36 ms	0.3
4	1	1	Φ_1	32×32	44	65 ms	6.6	76 ms	6.6
5	1	1	Φ_1	64×64	52	72 ms	6.6	105 ms	6.6
6	1	1	Φ_1	128×128	60	93 ms	6.6	101 ms	6.6
7	1	1	Φ_{12}	4×4	20	0 ms	0.3	1 ms	0.3
8	1	1	Φ_{12}	8×8	28	5 ms	0.3	6 ms	0.3
9	1	1	Φ_{12}	16×16	36	12 ms	0.3	13 ms	0.3
10	1	1	Φ_{12}	32×32	44	30 ms	6.6	21 ms	6.6
11	1	1	Φ_{12}	64×64	52	53 ms	6.6	28 ms	6.6
12	1	1	Φ_{12}	128×128	60	70 ms	6.6	40 ms	6.6
13	1	1	Φ_{13}	4×4	28	4 ms	0.3	1 ms	0.3
14	1	1	Φ_{13}	8×8	38	73 ms	6.8	4 ms	0.3
15	1	1	Φ_{13}	16×16	48	492 ms	6.8	19 ms	6.6
16	1	1	Φ_{13}	32×32	58	4 s	13.1	53 ms	6.6
17	1	1	Φ_{13}	64×64	68	36.2 s	19.4	303 ms	6.6
18	1	1	Φ_{13}	128×128	78	330.8 s	50.9	1.9 s	6.6
19	1	1	Φ	4×4	28	2 ms	0.3	0 ms	0.3
20	1	1	Φ	8×8	38	44 ms	6.8	5 ms	0.3
21	1	1	Φ	16×16	48	292 ms	6.8	17 ms	6.6
22	1	1	Φ	32×32	58	2.5 s	13.1	46 ms	6.6
23	1	1	Φ	64×64	68	22.4 s	19.4	182 ms	6.6
24	1	1	Φ	128×128	78	242.3 s	50.9	1.3 s	6.6
25	1	2	Φ_1	4×4	31	17 ms	6.6	11 ms	6.6
26	1	2	Φ_1	8×8	43	347 ms	6.6	248 ms	6.6
27	1	2	Φ_1	16×16	55	1.7 s	19.4	1.8 s	12.9
28	1	2	Φ_1	32×32	67	9.1 s	38.3	7.6 s	19.3
29	1	2	Φ_1	64×64	79	27.3 s	63.5	28.7 s	38.2
30	1	2	Φ_1	128×128	91	74.1 s	88.6	75.6 s	44.6
31	1	2	Φ_{12}	4×4	31	11 ms	6.6	19 ms	6.6
32	1	2	Φ_{12}	8×8	43	186 ms	6.6	398 ms	6.6
33	1	2	Φ_{12}	16×16	55	1.1 s	19.4	3.9 s	13
34	1	2	Φ_{12}	32×32	67	5.6 s	25.7	14.8 s	19.4
35	1	2	Φ_{12}	64×64	79	17.3 s	44.6	49.5 s	32
36	1	2	Φ_{12}	128×128	91	53.8 s	57.2	104.2 s	45
37	1	2	Φ_{13}	4×4	47	117 ms	6.6	14 ms	6.6
38	1	2	Φ_{13}	8×8	63	15 s	38.3	401 ms	6.6
39	1	2	Φ_{13}	16×16	79	14.9m	365.5	4.2 s	19.3
40	1	2	Φ_{13}	32×32	95	mem out	mem out	34.4 s	50.8
41	1	2	Φ	4×4	47	205 ms	6.8	40 ms	6.6
42	1	2	Φ	8×8	63	9.8 s	31.9	0.9 s	6.6
43	1	2	Φ	16×16	79	400.3 s	239.7	5.1 s	19.4
44	1	2	Φ	32×32	95	155.8 min	1209	33.1 s	38.3
45	1	3	Φ_1	4×4	42	488 ms	6.6	182 ms	6.6
46	1	3	Φ_1	8×8	58	108.7 s	176.7	17.5 s	31.9
47	1	3	Φ_1	16×16	74	38.6 min	1391.5	181.9 s	201.8
48	1	3	Φ_1	32×32	90	mem out	mem out	15 min	485.1
49	1	3	Φ_{12}	4×4	42	417 ms	6.6	301 ms	6.6
50	1	3	Φ_{12}	8×8	58	25.8 s	82.2	13.6 s	31.8
51	1	3	Φ_{12}	16×16	74	9.6 min	648.7	168.1 s	271
52	1	3	Φ_{12}	32×32	90	mem out	mem out	14.9 min	711.8
53	1	3	Φ_{13}	4×4	66	22 s	50.8	0.8 s	6.8
54	1	3	Φ_{13}	8×8	88	mem out	mem out	98.4 s	101.2
55	1	3	Φ	4×4	66	35.1 s	50.8	1.1 s	6.7
56	1	3	Φ	8×8	88	88.9 min	1227.8	28.4 s	44.5
57	1	4	Φ_1	4×4	53	65.5 s	107.5	15.8 s	25.8
58	1	4	Φ_1	8×8	73	mem out	mem out	91.8 min	1252.9
59	1	4	Φ_{12}	4×4	53	28.1 s	69.6	14.1 s	31.9
60	1	4	Φ_{12}	8×8	73	mem out	mem out	42.2 min	1278.1
61	1	4	Φ_{13}	4×4	85	181.1 min	1303.4	119.2 s	107.6
62	1	4	Φ	4×4	85	83.2 min	806.1	40.4 s	38.3

Table 5.4: Evaluation of approaches on a robot motion planning case study with perfect agents

						Centralized		Compositional	
Ex. #	$\lvert\mathcal{M}^u\rvert$	$\lvert\mathcal{M}^c\rvert$	objective	size	$\lvert\mathcal{V}\rvert$	time	mem (MB)	time	mem (MB)
63	2	1	Φ_1	4×4	29	34 ms	0.3	1 ms	0.3
64	2	1	Φ_1	8×8	41	73 ms	6.8	33 ms	6.6
65	2	1	Φ_1	16×16	53	147 ms	6.7	143 ms	6.6
66	2	1	Φ_1	32×32	65	254 ms	6.6	0.7 s	6.6
67	2	1	Φ_1	64×64	77	353 ms	6.6	1.2 s	6.6
68	2	1	Φ_1	128×128	89	1.1 s	6.6	2.2 s	6.7
69	2	1	Φ_{12}	4×4	29	1 ms	0.3	1 ms	0.3
70	2	1	Φ_{12}	8×8	41	18 ms	6.8	29 ms	6.6
71	2	1	Φ_{12}	16×16	53	76 ms	6.7	142 ms	6.6
72	2	1	Φ_{12}	32×32	65	140 ms	6.6	378 ms	6.6
73	2	1	Φ_{12}	64×64	77	0.6 s	6.6	1.1 s	6.6
74	2	1	Φ_{12}	128×128	89	0.7 s	6.6	2.2 s	6.7
75	2	1	Φ_{13}	4×4	37	46 ms	6.8	1 ms	0.3
76	2	1	Φ_{13}	8×8	51	112 s	322	31 ms	6.6
77	2	1	Φ_{13}	16×16	65	mem out	mem out	155 ms	6.6
78	2	1	Φ_{13}	32×32	79	mem out	mem out	429 ms	6.6
79	2	1	Φ_{13}	64×64	93	mem out	mem out	1.4 s	6.6
80	2	1	Φ_{13}	128×128	107	mem out	mem out	3.7 s	6.7
81	2	1	Φ	4×4	37	217 ms	6.8	2 ms	0.3
82	2	1	Φ	8×8	51	106.4 s	322	33 ms	6.6
83	2	1	Φ	16×16	65	mem out	mem out	155 ms	6.6
84	2	1	Φ	32×32	79	mem out	mem out	424 ms	6.6
85	2	1	Φ	64×64	93	mem out	mem out	1.4 s	6.6
86	2	1	Φ	128×128	107	mem out	mem out	3.5 s	6.7
87	2	2	Φ_1	4×4	40	97 ms	6.7	68 ms	6.6
88	2	2	Φ_1	8×8	56	5.4 s	25.7	4.4 s	19.2
89	2	2	Φ_{12}	4×4	40	84 ms	6.7	195 ms	6.6
90	2	2	Φ_{12}	8×8	56	4.5 s	19.4	13.5 s	19.4
91	2	2	Φ_{13}	4×4	56	3.2 s	25.7	113 ms	6.6
92	2	2	Φ_{13}	8×8	76	18.7 min	629.9	6.2 s	25.6
93	2	2	Φ	4×4	56	3.2 s	19.4	201 ms	6.6
94	2	2	Φ	8×8	76	10.6 min	460	14.4 s	19.4
95	2	3	Φ_1	4×4	51	9.7 s	32	3.3 s	12.9
96	2	3	Φ_1	8×8	71	53.8 min	1423	421.4 s	264.8
97	2	3	Φ_{12}	4×4	51	5.8 s	25.6	3.2 s	13.1
98	2	3	Φ_{12}	8×8	71	14.2 min	642.4	481.3 s	277.4
99	2	3	Φ_{13}	4×4	75	19.1 min	497.8	8.4 s	25.9
100	2	3	Φ_{13}	8×8	101	mem out	mem out	30.2 min	800.2
101	2	3	Φ	4×4	75	time out	time out	5 s	13.3
102	2	3	Φ	8×8	101	mem out	mem out	12.7 min	302.6
103	2	4	Φ_1	4×4	62	mem out	mem out	385.7 s	158

Table 5.5: Evaluation of approaches on a robot motion planning case study with imperfect agents

						Centralized		Compositional	
Ex. #	$\lvert\mathcal{M}^u\rvert$	$\lvert\mathcal{M}^c\rvert$	objective	size	$\lvert\mathcal{V}\rvert$	time	mem (MB)	time	mem (MB)
$\lvert\mathcal{M}^u\rvert$	$\lvert\mathcal{M}^c\rvert$	objective	size	$\lvert\mathcal{V}\rvert$	time	mem (MB)	time	mem (MB)	
104	1	1	Φ_1	4×4	68	115 ms	0.3	142 ms	0.3
105	1	1	Φ_1	5×5	124	153 ms	6.7	210 ms	6.7
106	1	1	Φ_1	6×6	124	186 ms	6.7	201 ms	6.7
107	1	1	Φ_1	7×7	124	314 ms	6.7	248 ms	6.7
108	1	1	Φ_1	8×8	124	363 ms	7	336 ms	7
109	1	1	Φ_{12}	4×4	68	17 ms	0.3	16 ms	0.3
110	1	1	Φ_{12}	5×5	124	61 ms	6.7	50 ms	6.7
111	1	1	Φ_{12}	6×6	124	110 ms	6.7	115 ms	6.7
112	1	1	Φ_{12}	7×7	124	173 ms	6.7	241 ms	6.7
113	1	1	Φ_{12}	8×8	124	215 ms	7	239 ms	7
114	1	1	Φ_{13}	4×4	76	15 ms	0.4	26 ms	0.4
115	1	1	Φ_{13}	5×5	134	54 ms	6.7	57 ms	6.7
116	1	1	Φ_{13}	6×6	134	101 ms	6.7	116 ms	6.7
117	1	1	Φ_{13}	7×7	134	173 ms	6.7	206 ms	6.7
118	1	1	Φ_{13}	8×8	134	246 ms	7	381 ms	7
119	1	1	Φ	4×4	76	14 ms	0.4	14 ms	0.4
120	1	1	Φ	5×5	134	76 ms	6.7	57 ms	6.7
121	1	1	Φ	6×6	134	150 ms	6.7	114 ms	6.7
122	1	1	Φ	7×7	134	180 ms	6.7	253 ms	6.7
123	1	1	Φ	8×8	134	299 ms	7	428 ms	7
124	1	1	Φ_1	9×9	196	1.2 s	6.7	1.3 s	6.7
125	1	1	Φ_1	10×10	196	1.2 s	6.7	1.1 s	6.7
126	1	1	Φ_{12}	9×9	196	0.6 s	6.7	0.9 s	6.7
127	1	1	Φ_{12}	10×10	196	0.9 s	6.7	1.1 s	6.7
128	1	1	Φ_{13}	9×9	206	0.8 s	6.7	1.1 s	6.7
129	1	1	Φ_{13}	10×10	206	1.5 s	6.7	1.3 s	6.7
130	1	1	Φ	9×9	206	0.9 s	6.7	1.1 s	6.7
131	1	1	Φ	10×10	206	1.1 s	6.7	1.2 s	6.7

Table 5.6: Evaluation of approaches on a robot motion planning case study with imperfect agents

						Centralized		Compositional	
Ex. #	$\lvert\mathcal{M}^u\rvert$	$\lvert\mathcal{M}^c\rvert$	objective	size	$\lvert\mathcal{V}\rvert$	time	mem (MB)	time	mem (MB)
$\lvert\mathcal{M}^u\rvert$	$\lvert\mathcal{M}^c\rvert$	objective	size	$\lvert\mathcal{V}\rvert$	time	mem (MB)	time	mem (MB)	
132	1	2	Φ_1	4×4	127	1.3 s	6.7	189 ms	6.7
133	1	2	Φ_1	5×5	235	7.7 s	13	1.3 s	6.7
134	1	2	Φ_1	6×6	235	25.2 s	31.9	4.9 s	13
135	1	2	Φ_1	7×7	235	84.2 s	69.7	16.4 s	25.6
136	1	2	Φ_1	8×8	235	242.6 s	126.3	37 s	44.7
137	1	2	Φ_{12}	4×4	127	1.7 s	6.7	0.6 s	6.7
138	1	2	Φ_{12}	5×5	235	8.4 s	13	2.3 s	6.7
139	1	2	Φ_{12}	6×6	235	28.6 s	31.9	10.2 s	19.3
140	1	2	Φ_{12}	7×7	235	83.5 s	69.7	35 s	31.9
141	1	2	Φ_{12}	8×8	235	229.7 s	126.6	95 s	57.1
142	1	2	Φ_{13}	4×4	143	1.4 s	6.7	303 ms	6.7
143	1	2	Φ_{13}	5×5	255	10 s	25.6	1.3 s	6.7
144	1	2	Φ_{13}	6×6	255	38.2 s	57.1	5 s	13
145	1	2	Φ_{13}	7×7	255	159.5 s	132.6	16.7 s	25.6
146	1	2	Φ_{13}	8×8	255	8.9 min	252.2	38.3 s	51
147	1	2	Φ	4×4	143	2.3 s	6.7	0.7 s	6.7
148	1	2	Φ	5×5	255	10.9 s	25.6	2.5 s	13
149	1	2	Φ	6×6	255	46.2 s	50.8	10 s	19.3
150	1	2	Φ	7×7	255	123.7 s	107.4	54.4 s	31.9
151	1	2	Φ	8×8	255	344.5 s	202.1	129.9 s	57.1
152	1	2	Φ_1	9×9	375	27.8 min	390.7	113.2 s	82.3
153	1	2	Φ_1	10×10	375	time out	time out	256.2 s	151.5
154	1	2	Φ_{12}	9×9	375	time out	time out	306 s	94.9
155	1	2	Φ_{12}	10×10	375	time out	time out	9.7 min	176.7
156	1	2	Φ_{13}	9×9	395	time out	time out	114.9 s	88.6
157	1	2	Φ_{13}	10×10	395	time out	time out	279.9 s	157.8
158	1	2	Φ	9×9	395	time out	time out	309.9 s	101.2
159	1	2	Φ	10×10	395	time out	time out	9.6 min	176.7
160	1	3	Φ_1	4×4	186	144.3 s	69.7	0.9 s	6.7
161	1	3	Φ_1	5×5	346	time out	time out	4.5 s	19.3
162	1	3	Φ_1	6×6	346	time out	time out	17.7 s	38.2
163	1	3	Φ_1	7×7	346	time out	time out	65 s	82.3
164	1	3	Φ_1	8×8	346	time out	time out	190.9 s	176.7
165	1	3	Φ_1	9×9	554	time out	time out	9.3 min	346.7
166	1	3	Φ_1	10×10	554	time out	time out	24.6 min	730.6
167	1	3	Φ_{12}	4×4	186	145.3 s	63.4	1 s	6.7
168	1	3	Φ_{12}	5×5	346	time out	time out	5.8 s	19.3
169	1	3	Φ_{12}	6×6	346	time out	time out	24.5 s	44.5
170	1	3	Φ_{12}	7×7	346	time out	time out	89.5 s	94.9
171	1	3	Φ_{12}	8×8	346	time out	time out	280.5 s	195.6
172	1	3	Φ_{12}	9×9	554	time out	time out	13.2 min	378.1
173	1	3	Φ_{12}	10×10	554	time out	time out	31.4 min	768.4
174	1	3	Φ_{13}	4×4	210	265.8 s	214.5	0.9 s	6.7
175	1	3	Φ_{13}	5×5	376	time out	time out	8.3 s	19.4
176	1	3	Φ_{13}	6×6	376	time out	time out	49.2 s	57.1
177	1	3	Φ_{13}	7×7	376	time out	time out	260.4 s	132.7
178	1	3	Φ_{13}	8×8	376	time out	time out	483.9 s	214.5
179	1	3	Φ_{13}	9×9	584	time out	time out	31.7 min	441.1
180	1	3	Φ	5×5	376	time out	time out	8 s	25.7
181	1	3	Φ	6×6	376	time out	time out	36 s	50.8
182	1	3	Φ	7×7	376	time out	time out	129.5 s	101.2
183	1	3	Φ	8×8	376	time out	time out	343.4 s	201.9
184	1	3	Φ	9×9	584	time out	time out	16.5 min	378.2
185	1	3	Φ	10×10	584	time out	time out	39.6 min	774.7

6

Related Work

The synthesis problem was first recognized by Church [Chu62]. The problem of synthesizing reactive systems from a specification given in linear temporal logic was considered by Pnueli et al. [PR89], where they propose a synthesis algorithm that first transforms the LTL specification into a Büchi automaton, which is then translated into a deterministic Rabin automaton using Safra's determinization procedure [Saf88]. This double translation causes a doubly exponential time complexity which is unavoidable [Ros92]. The high complexity of the synthesis process was discouraging, however, it was shown later that there are several interesting cases where the synthesis problem can be solved in polynomial time [ALT04, PAMS98]. Bloem et al. [BJP$^+$12] present polynomial time algorithms for the realizability and synthesis problems for a more general fragment of LTL known as Generalized Reactivity (1) (GR(1)). They show the efficiency and expressivity of GR(1) by applying their algorithms to a realistic industrial hardware case study of a medium size. We also consider GR(1) specifications in Chapter 3 and show how they can be automatially refined by synthesizing additional assumptions-guarantees using patterns.

Compositional reactive synthesis has been considered in some recent works. Kupferman et al. [KV05] propose a Safraless approach that reduces the LTL realizability problem to Büchi games. Their approach is then extended to treat specifications that are conjunction of LTL properties compositionally [KPV06]. There is no notion of maximally permissive strategy for büchi games, and to our best knowledge their algorithms are not implemented. Baier et al. [BKK11] give a compositional framework for treating multiple linear-time objectives inductively. To this end, they introduce the concept of most general strategies which generate all decision functions that guarantee the objective under consideration. Sohail et al. [SS09] propose an algorithm to compositionally construct a parity game from

conjunctive LTL specifications. Filiot et al. [FJR11] present monolithic and compositional algorithms to solve the LTL realizability problem. They reduce the LTL realizability problem to solving safety games, and show that for LTL specifications written as conjunction of smaller LTL formulas, the problem can be solved compositionally by first computing winning strategies for each conjunct. Moreover, they show that compositional algorithms can handle fairly large LTL specifications.

Two-player games of imperfect information are studied in [Rei84, CH05, DWDR06, CDHR06], and it is shown that they are often more complicated than games of perfect information. The algorithmic difference is exponential, due to a subset construction that turns a game of imperfect information into an equivalent game of perfect information. In Chapter 5, we build on the results of [FJR11, CDHR06] and extend and adapt their methods to treat multi-agent systems with imperfect agents. To the best of our knowledge, compositional reactive synthesis is not studied in the context of multi-agent systems and robot motion planning.

The setting considered in Chapter 4 is different from the ones in [KV05, BKK11, SS09, FJR11] as we are interested in synthesizing from a library of controllers that can be reused. The problem of LTL synthesis from a library of reusable components is also considered in [LV13]. Sequential composition of controllers considered in Chapter 4 is similar to control-flow composition in [LV13] and is inspired by software systems. Although by enumerating the parameter values and instantiating parametric controllers to obtain a library of non-parametric controllers our problem can be reduced to the one considered in [LV13], such naive enumeration may lead to an exponentially larger number of controllers in the library, making the method infeasible in practice. Our algorithms *symbolically* explore the parametric space, thus avoiding the excessive explicit enumeration. To the best of our knowledge, there is no implementation of the methods proposed in [LV13].

The synthesis problem for distributed reactive was first considered by Pnueli et al. [PR90]. They showed that distributed synthesis problem is undecidable in general and has non-elementary complexity for pipeline architectures. Kupferman et al. [KV01] propose a automata-based synthesis algorithm for pipeline and ring architectures in which information flows in either one or both directions. Mohalik et al. [MW03] provide an alternative game-theoretical approach. Madhusudan et al. [MT01] consider the special case of synthesizing distributed controllers for reactive systems against *local* specifications

(each property only refers to the variables of one of the processes), and show that a larger class of architectures become decidable in comparison to the analogous problem for global specifications. Finkbeiner et al. [FS05] show that the distributed synthesis problem is decidable if and only if the architecture does not contain an *information fork*. Madhusudan et al. [MT02] consider the problem of synthesizing controllers for distributed asynchronous systems and show that under some severe restrictions the problem is decidable. Schewe et al. [SF07b] show that the synthesis of asynchronous distributed systems is decidable if and only if at most one process implementation is unknown. Finkbeiner et al. [FS13] introduce the bounded synthesis approach where a bound on a system parameter such as the number of states is fixed a priori, and only those implementations that fall below the bound are considered. The main idea behind their solution is a translation from linear temporal logic specifications to sequences of safety tree automata which underapproximate the specification and eventually become empty-equivalent. They show that bounded synthesis is applicable to variations of synthesis problem including synchronous and asynchronous distributed systems. Bounded synthesis approach is implemented symbolically using binary decision diagrams in [Ehl12, Ehl11] and using anti-chains in [FJR09].

The problem of correcting an unrealizable LTL specification by constructing an additional environment assumption is also studied by [CHJ08]. They give an algorithm for computing the assumption which only constrains the environment and is as weak as possible. Their approach is more general than the counter-strategy-guided refinement approach proposed in Chapter 3 as they consider general LTL specifications. However, the synthesized assumption is a Büchi automaton that might not translate to an LTL formula and can be difficult for the user to understand. Moreover, the resulting specification is not necessarily compatible with the design intent. The closest work to ours is the work by Li et al. [LDS11] where they propose a template-based specification mining approach to find additional assumptions on the environment that can be used to rule out the counter-strategy. A template is an LTL formula with at least one placeholder, $?_b$, that can be instantiated by the Boolean variable b or its negation. Templates are used to impose a particular structure on the form of generated candidates and are engineered by the user based on her knowledge of the environment. A set of candidate assumptions is generated by enumerating all possible instantiations of the defined templates. For a given counter-strategy, their method finds an assumption from the set of candidate assumptions which is satisfied by the counter-strategy. By adding the

negation of such an assumption to the specification, they remove the behavior described by the counter-strategy from the environment. This process is repeated until either the resulting specification becomes realizable or there is no candidate that is satisfied by the counter-strategy. Similar to their work, we consider unrealizable GR(1) specifications and achieve realizability by adding environment assumptions to the specification. But, unlike them, we directly work on the counter-strategies to synthesize a set of candidate assumptions that can be used to rule out the counter-strategy. Similar to templates, patterns impose structure on the assumptions. However, our method synthesizes the patterns based on the counter-strategy and the user does not need to manipulate them. We only require the user to specify a subset of variables to be used in the search for missing assumptions. The user can specify a subset that she thinks leads to the unrealizability. In our method, the maximum number of generated assumptions for a given counter-strategy is independent from what subset of variables is considered, whereas increasing the size of the chosen subset of variables in [LDS11] will result in exponential growth in the number of candidates, while only a small number of them might hold over all runs of the counter-strategy (unlike our method). Moreover, we compute the weakest environment assumptions for the considered structure and given subset of variables. Our work takes an initial step toward bridging the gap between [CHJ08] and [LDS11]. Our method synthesizes environment assumptions that are simple formulas, making them easy to understand and practical, and they also constrain the environment as weakly as possible within their structure.

Chatterjee et al. [CH07] consider assume-guarantee synthesis problem and show that it can be solved by computing secure-equilibrium strategies. The main idea is that processes are adversarial, but will not compete with each other at the price of violating their own specification. Bloem et al. [BCJK15] extend the assume-guarantee synthesis approach proposed in [CH07] for simultaneous synthesis of multiple processes with partial information restrictions. In Chapter 3, we use a different approach for assume-guarantee synthesis that is based on specification refinement through inferring LTL formulas from strategies and counter-strategies.

The controller synthesis problem for systems with multiple controllable agents from a high-level temporal logic specification is also considered in many recent works (e.g., [KGFP09, WTM12, KgWT11]). A common theme is based on first computing a discrete controller satisfying the LTL specification over a discrete abstraction of the system, which is

then used to synthesize continues controllers guaranteed to fulfill the high-level specification. In many of these works (e.g., [WUB+13, KB10]) the agents' models are composed (either from the beginning or incrementally) to obtain a central model. The product of the central model with the specification automaton is then constructed and analyzed to compute a strategy. In [SRK+14], authors present a compositional motion planning framework for multi-robot systems based on a reduction to satisfiability modulo theories. However, their model cannot handle uncertain or dynamic environment. In [KGFP09, OTM11] it is proposed that systems with multiple components can be treated in a decentralized manner by considering one component as a part of the environment of another component. However, these reactive approaches cannot address the need for joint decision making and cooperative objectives. In Chapter 5 we consider compositional and symbolic algorithms for solving games in presence of a dynamic and possibly adversarial environment. Note that within the scope of this dissertation, we assumed that a finite-state abstraction of the system is given and we presented compositional algorithms for synthesizing *discrete* controllers. The computed controllers can then be refined to controllers enforcing the specification over the original system using the techniques in the literature [Tab09].

7

Conclusions and Future Work

This chapter concludes the dissertation by providing a brief summary of the proposed methods followed by some directions toward which they can be improved and extended. We considered the problem of automated synthesis of controllers for multi-agent systems from high-level temporal logic specifications. The key insight was to consider more restricted yet practically useful subclasses of the general problem. The overall theme of the solution approaches was to take advantage of the existing structure in systems in order to decompose the synthesis problem into smaller and more manageable subproblems, and to achieve more efficient synthesis algorithms through compositional synthesis techniques. We explored three different frameworks for compositional synthesis:

- We showed how automated refinement of specifications can be used to revise the specifications of the components in the context of compositional synthesis. We proposed three different approaches for compositional refinement of specifications. The choice of the appropriate approach depends on the size of the problem (e.g., number of states in strategies and counter-strategies) and the level of acceptable coupling between components. Supplying more information about the strategies of the components with realizable local specifications to unrealizable specification under refinement, reduces the number of scenarios the game solver needs to consider, and facilitates the synthesis procedure, while increasing the coupling between components. Overall, patterns provide a tool for the designer to refine and complete temporal logic specifications.
- We presented a framework for compositional and symbolic synthesis from a library of parametric and reactive controllers. We also showed how these controllers can be synthesized from parametric objectives specified by the user.
- We proposed a framework for controller synthesis for dynamically decoupled multi-

agent systems. We showed that, by taking advantage of the structure in the system to compositionally synthesize the controllers, and by representing and exploring the state space symbolically, we can achieve better scalability and solve more realistic problems. Our preliminary results show the potential of reactive synthesis as planning algorithms in the presence of dynamically changing and adversarial environment.

There are several directions in which the work described in this dissertation can be extended and improved upon. We summarize what we consider to be the most promissing directions next.

Refining specifications using counter-strategies. Counter-strategies provide useful information for explaining reasons for unrealizability. However, there can be multiple ways to rule out a counter-strategy. It remains to investigate how the multiplicity of the candidates generated by the methods proposed in Chapter 3 can be used to synthesize better refinements. Furthermore, the proposed methods ask the user for subsets of variables to be used in generating candidates. The choice of the subsets can significantly impact how fast the algorithms can find a refinement. Automatically finding good subsets of variables that contribute to the unrealizability problem is another future direction.

Computing patterns for LTL specifications. The problem of inferring formulas from transition systems has a close relationship with model checking. In model checking, given a transition system and a specification in a formal language such as LTL, the goal is to decide whether the specification is satisfied over all possible executions of the transition system. On the other hand, given a transition system and a *temporal template*, the goal of *pattern synthesis* is to compute formulas with the given temporal template that hold over all executions of the transition system. Intuitively, temporal templates are LTL formulas where some of the propositional parts are left-out for synthesis (similar to sketching [SL09] in program synthesis). We illustrate the problem with an example.

Consider the transition system shown in Figure 7.1 where a and b are two Boolean variables and states are labeled by propositional formulas which hold in them. Assume the user gives $\square((\neg a \wedge b) \rightarrow \Diamond ?_{a,b})$ as a temporal template. Intuitively, the user wants to know what is guaranteed to happen eventually after any visit to a state satisfying the propositional formula $\neg a \wedge b$. Unknown propositional formula $?_{a,b}$ is to be synthesized and replaced by a propositional formula over a and b. For example, the synthesis process may compute $?_{a,b} = (a \wedge \neg b) \vee (a \wedge b) = a$, leading to the LTL formula $\square((\neg a \wedge b) \rightarrow \Diamond a)$ which

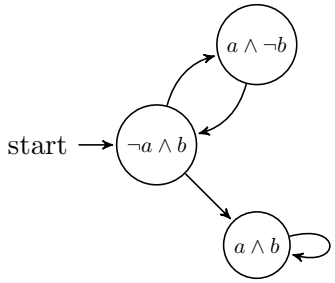


Figure 7.1: Example of a transition system

is satisfied over all runs of the given transition system.

In the framework proposed in Chapter 3, we only consider temporal templates with forms allowed in GR(1) fragment of the linear temporal logic. However, LTL has more expressive power and hence some properties are easier to express as LTL formulas. Extending the algorithms to be able to infer LTL formulas from strategies and counter-strategies and refine LTL specifications is one of the possible future directions.

Cooperation and competition between agents. In multi-agent systems we consider within the scope of this dissertation, the agents are either cooperative or adversarial. An interesting future direction is to extend this model and allow the user to specify which agents can cooperate to fulfill a specific objective. For example, consider a system with two controllable agents a_1 and a_2, and assume that the objectives of the system are for a_1 and a_2 to avoid collision while maintaining a maximum distance (e.g., for communication purposes). Assume agent a_1 can cooperate with a_2 to avoid collision, while it cannot trust a_2 with respect to maximum distance objective, i.e., in order for the system to satisfy the distance objective, a_1 must react to possible actions of agent a_2. Such specifications can also be expressed in alternating temporal logic [AHK02], however, their application to multi-agent systems and robot motion planning is not investigated. We are particularly interested in how methods presented in this dissertation can be extended to handle such objectives.

Robot motion planning case study. Another future direction is to integrate and apply the methods proposed in this dissertation to a more realistic robot motion planning case study with multiple controlled robots in presence of uncertain and dynamic environment, e.g., uncontrolled dynamic obstacles. The goal is to automatically synthesize controllers for the controlled agents from a high-level specification that can be deployed on a robotic platform or be simulated in a robotic simulation environment (e.g., Gazebo [KH04]). For

example, we can consider a case study similar to that of [SRK+14] where a group of controlled quadrotors are required to navigate through the room while avoiding collision with each other, maintaining formation and satisfy proximity constraints. Furthermore, in this case study there may exist uncontrolled and dynamic obstacles (e.g., uncontrolled quadrotors) and the synthesis process must take the uncertainty in the environment into account. The user specifies a high-level specification and the output will be low-level controllers that can be deployed on the quadrotors or simulated in a simulation environment. Such a case study, in addition to demonstrating the potential application of reactive synthesis in controller synthesis for multi-agent systems, can reveal possible challenges in interfacing between different layers of controller design, and provide some insight into more efficient solution approaches.

Modeling uncertainty about the environment. In the compositional synthesis framework proposed in Chapter 4, we assumed that the controllers have perfect information about the state of the system at any time-step. However, in practice, this assumption might be unrealistic, e.g., due to the imperfection and limitations of the sensors of the system. In future, we plan to investigate how our approach can be generalized to synthesize strategies for systems from a library of controllers with *partial* information.

Game structures of imperfect information provide a natural way for modeling systems that interact with partially observable and dynamic environment. Unfortunately, the scalability becomes a bigger issue due to the subset construction procedure that can lead to an exponentially larger perfect information game structure. In our implementation of the framework proposed in Chapter 5, we performed the subset construction procedure symbolically and we only constructed the part of it that is reachable from the initial state. One of our observations was that, by considering more structured observation functions for game structures of imperfect information, such as the ones considered in our case study where the robots show a "local" observation behavior, the worst case exponential blow-up in the constructed knowledge game structure does not occur in practice. In future, we plan to investigate how considering more restricted yet practical observation functions can enable us to handle systems with imperfect agents of larger size.

Bibliography

[AHK02] Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, 2002. [Cited on pages 86 and 110.]

[AKK11] Nora Ayanian, Vinutha Kallem, and Vijay Kumar. Synthesis of feedback controllers for multiple aerial robots with geometric constraints. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3126–3131. IEEE, 2011. [Cited on page 94.]

[ALT04] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for ltl fragments. *ACM Transactions on Computational Logic (TOCL)*, 5(1):1–25, 2004. [Cited on page 103.]

[AMT] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional synthesis of reactive controllers for multi-agent systems. [Cited on page 7.]

[AMT13] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of gr (1) temporal logic specifications. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 26–33. IEEE, 2013. [Cited on page 7.]

[AMT15] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Pattern-based refinement of assume-guarantee specifications in reactive synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 501–516. Springer, 2015. [Cited on page 7.]

[AMT16] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional synthesis with parametric reactive controllers. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 215–224. ACM, 2016. [Cited on page 7.]

[BA98] Tucker Balch and Ronald C Arkin. Behavior-based formation control for multirobot teams. *IEEE Transactions on Robotics and Automation*, 14(6):926–939, 1998. [Cited on page 1.]

[BCG+10] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy–a new requirements analysis tool with synthesis. In *Computer Aided Verification*, pages 425–429. Springer, 2010. [Cited on pages 39, 42, 43, and 52.]

[BCJK15] Roderick Bloem, Krishnendu Chatterjee, Swen Jacobs, and Robert Könighofer. Assume-guarantee synthesis for concurrent reactive programs with partial information. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 517–532. Springer, 2015. [Cited on page 106.]

[BGS06] Roderick Bloem, Harold N Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *Formal Methods in System Design*, 28(1):37–56, 2006. [Cited on pages 67, 69, and 94.]

[BJP+12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012. [Cited on pages 2, 20, 43, 44, 46, 84, and 103.]

[BKK11] Christel Baier, Joachim Klein, and Sascha Klüppelholz. A compositional framework for controller synthesis. In *Concurrency Theory (CONCUR)*, pages 512–527. Springer, 2011. [Cited on pages 103 and 104.]

[CDHR06] Krishnendu Chatterjee, Laurent Doyen, Thomas A Henzinger, and Jean-François Raskin. Algorithms for omega-regular games with imperfect information. In *Computer Science Logic*, pages 287–302. Springer, 2006. [Cited on pages 14, 16, and 104.]

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999. [Cited on pages 10 and 60.]

[CH05] Krishnendu Chatterjee and Thomas A Henzinger. Semiperfect-information games. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 1–18. Springer, 2005. [Cited on page 104.]

[CH07] Krishnendu Chatterjee and Thomas A Henzinger. Assume-guarantee synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 261–275. Springer, 2007. [Cited on page 106.]

[CHJ08] Krishnendu Chatterjee, Thomas A Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *Concurrency Theory (CONCUR)*, pages 147–161. Springer, 2008. [Cited on pages 38, 105, and 106.]

[Chu62] Alonzo Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, pages 23–35, 1962. [Cited on page 103.]

[DM06] William B Dunbar and Richard M Murray. Distributed receding horizon control for multi-vehicle formation stabilization. *Automatica*, 42(4):549–558, 2006. [Cited on page 84.]

[DWDR06] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. A lattice theory for solving games of imperfect information. In *Hybrid Systems: Computation and Control*, pages 153–168. Springer, 2006. [Cited on pages 14 and 104.]

[Ehl11] Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 272–275. Springer, 2011. [Cited on page 105.]

[Ehl12] Rüdiger Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012. [Cited on pages 18, 91, and 105.]

[FBKT00] Dieter Fox, Wolfram Burgard, Hannes Kruppa, and Sebastian Thrun. A probabilistic approach to collaborative multi-robot localization. *Autonomous Robots*, 8(3):325–344, 2000. [Cited on page 1.]

[FDF05] Emilio Frazzoli, Munther A Dahleh, and Eric Feron. Maneuver-based motion planning for nonlinear systems with symmetries. *IEEE Transactions on Robotics*, 21(6):1077–1091, 2005. [Cited on pages 56, 57, 58, and 59.]

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for LTL realizability. In *Computer Aided Verification*, pages 263–277. Springer, 2009. [Cited on page 105.]

[FJR11] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011. [Cited on pages 6, 18, 85, 91, 94, and 104.]

[FS05] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 321–330. IEEE, 2005. [Cited on page 105.]

[FS13] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013. [Cited on page 105.]

[GH82] Yuri Gurevich and Leo Harrington. Trees, automata, and games. In *Proceedings of the Fourteenth annual ACM Symposium on Theory of Computing*, pages 60–65. ACM, 1982. [Cited on page 16.]

[HLW00] Dan Halperin, J-C Latombe, and Randall H Wilson. A general framework for assembly planning: The motion space approach. *Algorithmica*, 26(3-4):577–601, 2000. [Cited on page 1.]

[JWE97] James S Jennings, Greg Whelan, and William F Evans. Cooperative search and rescue with a team of mobile robots. In *8th International Conference on Advanced Robotics (ICAR)*, pages 193–200. IEEE, 1997. [Cited on page 1.]

[KB10] Marius Kloetzer and Calin Belta. Automatic deployment of distributed teams of robots from temporal logic motion specifications. *IEEE Transactions on Robotics*, 26(1):48–61, 2010. [Cited on page 107.]

[KBB06] Tamás Keviczky, Francesco Borrelli, and Gary J Balas. Decentralized receding horizon control for large scale dynamically decoupled systems. *Automatica*, 42(12):2105–2115, 2006. [Cited on page 84.]

[KGFP09] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009. [Cited on pages 94, 106, and 107.]

[KgWT11] Hadas Kress-gazit, Tichakorn Wongpiromsarn, and Ufuk Topcu. Correct, reactive robot control from abstraction and temporal logic specifications, 2011. [Cited on page 106.]

[KH04] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154. IEEE, 2004. [Cited on page 110.]

[KHB09] Robert Konighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 152–159. IEEE, 2009. [Cited on pages 38, 39, and 42.]

[KPV06] Orna Kupferman, Nir Piterman, and Moshe Vardi. Safraless compositional synthesis. In *Computer Aided Verification (CAV)*, pages 31–44. Springer, 2006. [Cited on pages 39 and 103.]

[KV01] Orna Kupferman and Moshe Vardi. Synthesizing distributed systems. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 389–398. IEEE, 2001. [Cited on page 104.]

[KV05] Orna Kupferman and Moshe Y Vardi. Safraless decision procedures. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 531–540. IEEE, 2005. [Cited on pages 103 and 104.]

[LaV06] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006. [Cited on page 24.]

[LDS11] Wenchao Li, Lili Dworkin, and Sanjit A Seshia. Mining assumptions for synthesis. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 43–50. IEEE, 2011. [Cited on pages 46, 49, 105, and 106.]

[LV13] Yoad Lustig and Moshe Y Vardi. Synthesis from component libraries. *International Journal on Software Tools for Technology Transfer*, 15(5-6):603–618, 2013. [Cited on pages 57, 59, and 104.]

[McM] K. McMillan. Cadence SMV. http://www.kenmcmil.com/smv.html. [Cited on pages 44 and 52.]

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS 95*, pages 229–242. Springer, 1995. [Cited on pages 17, 66, and 69.]

[MT01] Parthasarathy Madhusudan and PS Thiagarajan. Distributed controller synthesis for local specifications. In *Automata, languages and programming*, pages 396–407. Springer, 2001. [Cited on page 104.]

[MT02] Parthasarathy Madhusudan and PS Thiagarajan. A decidable class of asynchronous distributed controllers. In *Concurrency Theory (CONCUR)*, pages 145–160. Springer, 2002. [Cited on page 105.]

[MW03] Swarup Mohalik and Igor Walukiewicz. Distributed games. In *Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, pages 338–351. Springer, 2003. [Cited on page 104.]

[OTM11] Necmiye Ozay, Ufuk Topcu, and Richard M Murray. Distributed power allocation for vehicle management systems. In *50th IEEE Conference on Decision and Control and European Control Conference*, pages 4841–4848. IEEE, 2011. [Cited on pages 20, 21, 22, 39, and 107.]

[PAMS98] Amir Pnueli, Eugene Asarin, Oded Maler, and Joseph Sifakis. Controller synthesis for timed automata. In *Proceedings of System Structure and Control. Elsevier*. Citeseer, 1998. [Cited on page 103.]

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM symposium on Principles of programming languages*, pages 179–190. ACM, 1989. [Cited on pages 2, 84, and 103.]

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, pages 746–757. IEEE, 1990. [Cited on pages 84 and 104.]

[PRA01] Gary Peterson, John Reif, and Salman Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41(7):957–992, 2001. [Cited on pages 84 and 87.]

[PSZ10] Amir Pnueli, Yaniv Sa'ar, and Lenore D Zuck. Jtlv: A framework for developing verification algorithms. In *Computer Aided Verification*, pages 171–174. Springer, 2010. [Cited on page 52.]

[RA10] Samuel Rodriguez and Nancy M Amato. Behavior-based evacuation planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 350–355. IEEE, 2010. [Cited on page 1.]

[RDJ95] Daniela Rus, Bruce Donald, and Jim Jennings. Moving furniture with teams of autonomous robots. In *Proceedings of 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 235–242. IEEE, 1995. [Cited on page 1.]

[Rei84] John H Reif. The complexity of two-player games of incomplete information. *Journal of computer and system sciences*, 29(2):274–301, 1984. [Cited on pages 16 and 104.]

[Ros92] Roni Rosner. *Modular synthesis of reactive systems*. PhD thesis, PhD thesis, Weizmann Institute of Science, 1992. [Cited on pages 20 and 103.]

[Saf88] Shmuel Safra. On the complexity of ω-automata. In *29th Annual Symposium on Foundations of Computer Science*, pages 319–327. IEEE, 1988. [Cited on page 103.]

[SF07a] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *Automated Technology for Verification and Analysis*, pages 474–488. Springer, 2007. [Cited on page 18.]

[SF07b] Sven Schewe and Bernd Finkbeiner. Synthesis of asynchronous systems. In *Logic-Based Program Synthesis and Transformation*, pages 127–142. Springer, 2007. [Cited on page 105.]

[ŞGBT08] Erol Şahin, Sertan Girgin, Levent Bayindir, and Ali Emre Turgut. Swarm robotics. In *Swarm intelligence*, pages 87–100. Springer, 2008. [Cited on page 84.]

[SL09] Armando Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems*, pages 4–13. Springer, 2009. [Cited on page 109.]

[SRK+14] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J Pappas, and Sanjit A Seshia. Automated composition of motion primitives for multi-robot systems from safe LTL specifications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1525–1532. IEEE, 2014. [Cited on pages 57, 58, 59, 87, 94, 107, and 111.]

[SS09] Saqib Sohail and Fabio Somenzi. Safety first: A two-stage algorithm for LTL games. In *Formal Methods in Computer-Aided Design*, pages 77–84. IEEE, 2009. [Cited on pages 103 and 104.]

[STZ+12] Zhiguo Shi, Jun Tu, Qiao Zhang, Lei Liu, and Junming Wei. A survey of swarm robotics system. In *Advances in Swarm Intelligence*, pages 564–572. Springer, 2012. [Cited on page 84.]

[Tab09] Paulo Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009. [Cited on page 107.]

[Vah] Arash Vahidi. Jdd. http://javaddlib.sourceforge.net/jdd/index.html. [Cited on pages 82 and 95.]

[WTM12] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830, 2012. [Cited on page 106.]

[WUB+13] Tichakorn Wongpiromsarn, Alphan Ulusoy, Calin Belta, Emilio Frazzoli, and Daniela Rus. Incremental synthesis of control policies for heterogeneous multi-agent systems with linear temporal logic specifications. In *IEEE International Conference on Robotics and Automation*, pages 5011–5018. IEEE, 2013. [Cited on page 107.]