RICE UNIVERSITY

A Constraint-Based Approach to Reactive Task and Motion Planning

by

Yue Wang

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. Swarat Chaudhuri, Chair
Associate Professor of Computer Science

Dr. Lydia E. Kavraki
Noah Harding Professor of Computer Science and Bioengineering

Dr. Moshe Y. Vardi
Karen Ostrum George Distinguished Service Professor in Computational Engineering

Houston, Texas
August, 2015
A Constraint-Based Approach to Reactive Task and Motion Planning

by

Yue Wang

Abstract

This thesis presents a novel and scalable approach for Reactive Task and Motion Planning. We consider changing environments with uncontrollable agents, where the robot needs a policy to respond correctly in the infinite interaction with the environment. Our approach operates on task and motion domains that combine actions over discrete states with continuous, collision-free paths. We synthesize a policy by iteratively verifying and searching for a policy candidate. For efficient verification, we employ Satisfiability Modulo Theories (SMT) solvers using a new extension of proof rules for Temporal Property Verification. For efficient policy search, we apply domain-specific heuristics to generalize verification failures. Furthermore, the SMT solver enables quantitative specifications such as energy limits. We benchmark our policy synthesizer in a mobile manipulation domain, showing that our approach offers better scalability compared to a state-of-the-art robotic synthesis tool in the tested benchmarks and demonstrating order-of-magnitude speedup from our heuristics.
Acknowledgments

First I would like to thank my advisers, Professor Swarat Chaudhuri and Professor Lydia Kavraki, for their valuable support during my first two years in the graduate school. I benefited a lot from their invaluable professional advice on many aspects of academia and research. Moreover, I could not have finished this thesis without their encouragements and helpful feedbacks along the way.

I would also like to thank my committee member, Professor Moshe Vardi, for his insightful comments when I presented the work of this thesis for the first time in the CAVR group meeting. I also want to mention that I was really enjoyed when I took COMP 509 last year and I learned a lot of background knowledge for this thesis from this course.

I have been fortunate to collaborate academically with some of my friends, Srinivas, Mark, Sailesh, Keliang and Neil. Without the help of Srinivas, Mark and Sailesh, I could not even get started with this research project. I could not remember how many times I went to Keliang’s office and had a brainstorming with him when I got stuck with my project. And Neil, the man who knows everything, he helped me a lot for preparing this thesis defense. I’d like to thank them all for their willingness to listen to me.

Finally, I would like to thank the other members of the Physical and Biological Computing Group at Rice University, especially Ryan and Morteza for many fruitful discussions about this project.

This work was supported by NSF CCF 1514372, NSF CCF 1162076 and NSF IIS 1317849.
## Contents

Abstract .......................................................... ii
Acknowledgments ................................................... iii
List of Illustrations ............................................... vii

1 Introduction ....................................................... 1
   1.1 Overview .................................................... 1
   1.2 Related Work ............................................... 2
      1.2.1 Task and Motion Planning .............................. 2
      1.2.2 Program Synthesis ..................................... 3
      1.2.3 Games ................................................... 4
      1.2.4 Reactive Synthesis for Robotic Applications ........... 5
   1.3 Contributions ............................................... 6
   1.4 Thesis Structure ............................................ 7

2 Problem Formulation ............................................ 8
   2.1 Reactive Task and Motion Planning ......................... 8
   2.2 Concurrent Games ........................................... 11
   2.3 Example .................................................... 13

3 Architecture Of ROBOSYNTH for Reactive Tasks ............... 20
   3.1 Placement Graph Generator ................................. 21
   3.2 Constraint Generator ....................................... 24
      3.2.1 Preprocessing .......................................... 24
      3.2.2 Encoding ............................................... 26
3.3 Game Generator .................................................. 29

4 Proof Rules for Concurrent games .................................. 31
  4.1 Linear Temporal Logic ........................................... 31
  4.2 Proof rules of Different Games ................................. 33
    4.2.1 Safety Property: $\square safety(s)$ ......................... 34
    4.2.2 Liveness Property: $\diamond dst(s)$ ....................... 37
    4.2.3 General LTL Property: $\varphi(s)$ ......................... 40
  4.3 Assumption On Concurrent Games ............................... 46

5 Policy Synthesizer .................................................. 48
  5.1 Implementation Overview ...................................... 50
    5.1.1 Candidate Generation ....................................... 51
    5.1.2 Proof Verification .......................................... 52
    5.1.3 Generalization .............................................. 52
  5.2 Algorithm Discussion ........................................... 53

6 Experimental Results ............................................... 55
  6.1 Implementation and Experimental Setup ....................... 55
  6.2 Results ......................................................... 57
    6.2.1 Benchmark: Safety Property ............................... 57
    6.2.2 Benchmark: Liveness Property ............................. 59
    6.2.3 Benchmark: GR(1) Property ................................ 60
    6.2.4 Benchmark: Liveness Property with Quantitative Constraints 61
    6.2.5 Generalization ............................................... 61
    6.2.6 Simulation Evaluation ...................................... 62

7 Conclusions and Future Work ..................................... 67
Bibliography 70
Illustrations

2.1 A Kitchen Environment ............................. 13
2.2 PDDL Definition of the pickUp Action  ................... 15
2.3 Plan Outline for Example  2.3.1 ........................ 17

3.1 Overall Structure of Robosynth for RTMP  .................. 21
3.2 Placement Graph Example ............................. 24

4.1 Proof rule RULESAFE for a safety property □safe(s) ........ 35
4.2 Proof rule RULELIVENESS for a liveness property ◇dst(s) .... 37
4.3 Proof rule RULELTL for a general LTL property ϕ ............ 41
4.4 Safety of Continuous Executions .......................... 47

5.1 Policy Synthesizer ........................................ 51
5.2 Illustration for one counter-example returned by the verifier ...... 53

6.1 Performance Results for the Safety Property Benchmark ...... 58
6.2 Performance Results for the Liveness Property Benchmark .... 59
6.3 Performance Results for the GR(1) Property Benchmark ...... 60
6.4 Performance of our Policy Synthesizer with Energy Constraint ... 62
6.5 Performance of our Policy Synthesizer without Generalization ... 63
6.6 Simulation: Enough Initial Energy ............................ 64
6.7 Simulation: Visiting the Charge Region ........................ 65
Chapter 1

Introduction

1.1 Overview

Traditionally, robots are deployed to perform repetitive tasks in highly structured environments such as assembly lines in factories and it is feasible to pre-program robots for such repetitive tasks. Today robots are often required to safely and correctly operate in environments that involve other uncontrollable agents such as humans. In these scenarios, since environments are changing over time due to uncontrollable activities of other agents, robots must react to the changes of environments online to ensure safety and accomplish desired tasks. Thus, rather than pre-programming finite instructions or pre-computing a single, linear plan, robots need a policy that determines the response in the infinite interaction with a changing environment, while also guaranteeing safety and achieving task goals. The Reactive Task and Motion Planning (RTMP) problem considered in this thesis is about efficiently constructing such policies while taking into account all feasible behaviors of other uncontrollable agents in the environment, which requires information from both motion-level and task-level.

This thesis presents a novel constraint-based approach for the Reactive Task and Motion Planning (RTMP) problem, which is scalable and enables quantitative specifications through the use of symbolic constraints and Satisfiability Modulo Theories (SMT) solvers [1]. Reasoning over sets of states using symbolic methods is more efficient than an explicit enumeration of states [2]. Furthermore, SMT solvers provide an expressive and efficient formulation of quantitative constraints such as energy limits. We validate our policy synthesis approach in a mobile manipulation domain with
human-robot interaction, demonstrating improved performance and expressiveness over an alternate robotic policy synthesis method.

We model the \textit{Reactive Task and Motion Planning} (RTMP) problem as a two-player game between the robot and the environment, then we find a winning policy for the robot in this game. We synthesize the policy by iteratively cycling between policy verification and policy candidate search \cite{3}. To verify the policy, we extend the proof rules of \cite{4,5}, which provide compact, symbolic constraints for verification. Then, we generalize verification failures by finding similar states using domain-specific heuristics. We use these invalid-state examples to search for a new policy candidate. This iterative policy synthesis procedure converges either to a winning policy or a proof that no such policy exists.

\section*{1.2 Related Work}

This thesis applies techniques from program synthesis to the generation of reactive policies for robot task and motion domains. In this section, we discuss related work including the \textit{Task and Motion Planning} (TMP) problem, the techniques for program synthesis that we apply and extend, the concurrent game structure that we apply to model the RTMP problem and alternative approaches for robot policy synthesis.

\subsection*{1.2.1 Task and Motion Planning}

\textit{Task planning} \cite{6} solves the problem of automatically constructing a sequence of actions to accomplish a task, which has been studied for a long time in the AI community. Many planners such as FF \cite{7}, Fast Downward \cite{8} and Madagascar \cite{9} have been developed to solve the task planning problem efficiently. However, these task planners operate in a discrete abstract space, while for robotic applications, we also need to consider the kinematic and dynamic constraints of the robot and the geometric constraints of the continuous physical world. Thus, every action that the robot performs must be mapped to a continuous collision-free trajectory between two robot
configurations. For instance, consider the manipulation problems where configuration spaces are usually of very high degrees of freedom (DOF). A naive discretization used with these task planners will perform poorly, since it will result in a huge state space due to the discretization of the high DOF configuration spaces of manipulation problems. Computing collision-free trajectories of “A to B” motions in a continuous physical space is the classical *Motion planning* problem. In the robotics community, much work [10–14] has been done to effectively generate such trajectories for complex robots.

*Task and Motion Planning* (TMP) combines motion planning of collision-free paths and task reasoning over discrete-valued actions [15–20]. These previous TMP approaches [15–20] assume that the environment is static. In contrast, we consider in this thesis a changing environment such that we require not just a plan describing a single execution path, but instead a policy describing the online response to uncontrollable events.

### 1.2.2 Program Synthesis

Policy synthesis for robotics is closely related to program synthesis for general software; both require determining the correct response to a variety of possible events or inputs. The classical program synthesis solves the problem of discovering a program, which implements the required system, from user intent expressed in the form of some logical formulas [21]. In the reactive case, except for the system, there is another component called the *environment* over which the system has no control. *Reactive Synthesis* is the problem of synthesizing a system that can satisfy the user requirement against all feasible environment behaviors [22]. In the formal methods community, reactive synthesis has been studied for a long time [22–26].

Synthesizing policies and programs is computationally hard due to the large – and possibly infinite – space of states and response. We address this computational challenge and efficiently synthesize robot policies by applying and extending
the syntax-guided synthesis (SyGuS) [3] framework, which uses a syntactic template, i.e., a grammar of potential policies, to improve efficiency by reducing the search space. Our policy synthesizer is based on the method of Counterexample-Guided Inductive Synthesis (CEGIS) [27], which iteratively verifies and searches for policy candidates. Our results show that our application of the SyGuS formulation and CEGIS method to robot policy synthesis scales better for the tested mobile manipulation benchmarks than an alternate GR(1) approach [26] that has been applied in the robotics domain [28,29].

We create two extensions to the SyGuS framework and CEGIS approach in order to synthesize robot policies. First, we modify SyGuS to handle the physical interaction necessary in robotic domains. Existing SyGuS solvers [30–32] synthesize software that does not interact with the physical world. Our solver incorporates low-level geometric information from the motion planner to handle the physical constraints required by robots. Second, we introduce an additional generalization step in the CEGIS method that greatly improves scalability. In typical CEGIS approaches, when a counterexample is generated, only this specific counterexample is added to the set of examples that guides policy search. In contrast, our synthesizer generalizes the specific counterexample to a set of similar examples using domain-specific heuristics (see Section 5.1.3). This generalization reduces the necessary number of iterations, improving scalability.

1.2.3 Games

Game solving has a close connection to program synthesis, especially in reactive settings. Many program synthesis approaches [22, 26, 33, 34] successfully solve the synthesis problems by modeling them as games. Most of these approaches consider turn-based games where only one player makes a move at each round. However, for robotics applications, we should consider the more general concurrent game structure where in each round, the robot and the environment choose their moves independently.
and simultaneously, and the moves they choose determines the successor state for the current state. This is because for the turn-based game model, we must assume that the environment is not changing while the robot makes its move. This is clearly not a realistic assumption in practice. For example, consider a robot operating in an environment in the presence of humans. When the robot is performing an action, it is possible that the person in the environment is also moving. Assuming that the environment is not changing while the robot is performing an action might cause unsafe intermediate states (e.g., collisions with the person).

In this thesis, we represent the system of the robot and the environment as a concurrent game [33,35], where in each round, the robot and the environment choose their moves independently and simultaneously. We solve this game by extending the standard proof system for Temporal Properties Verification [4, 5]. Though our proof rules are similar to those in [5], our back-end synthesis engine is different because our domain – robotic mobile manipulation – is different from that of [5]. Moreover, our proof rules handle concurrent games – where the agents’ actions are simultaneous – while the proof rules presented in [5] are for turn-based games – where the agents’ actions occur one-after-another.

1.2.4 Reactive Synthesis for Robotic Applications

In recent years, there has been an increasing interest in synthesizing reactive policies for robots and hybrid systems [28, 29, 36-39]. These approaches consider the continuous, differential dynamics of the hybrid system, but do not incorporate fast, randomized path planning. In contrast, we focus on the mobile manipulation domain where high dimensionality makes, efficient, collision-free path planning crucial. Furthermore, these previous works typically perform a combinatorial search over the state space. In practice, the state space may be exponentially large, limiting scalability. To improve scalability, work such as [28, 29, 37] uses a restricted fragment of Linear Temporal Logic (LTL) called Generalized Reactivity(1) (GR(1)). GR(1)
synthesis [26] is polynomial in the size of the state space; however, this state space may still be exponential [40]. In order to reduce computational complexity of temporal logic synthesis, some approaches [28, 29, 37, 39] synthesize control strategies in a receding horizon manner. In contrast, we avoid expensive combinatorial search using symbolic methods based on Satisfiability Modulo Theories (SMT) solvers, improving the scalability of policy synthesis. Using an SMT solver, we can efficiently handle quantitative constraints such as energy limits. Moreover, our approach enables specification of partial task knowledge using plan outlines [41], which reduces the search space of possible plans. Thus, compared to previous reactive synthesis approaches for robotics, we focus on manipulation rather than continuous dynamics, employ both efficient randomized path planning and efficient symbolic methods, efficiently handle quantitative constraints, and enable partial task specification using plan outlines.

1.3 Contributions

The novel contributions of the work presented in this thesis are:

- We developed a constraint-based method for solving RTMP problems, which reduces RTMP problems to concurrent two-player games and apply proof rules as symbolic constraints for winning policies. Moreover, compared to the existing temporal logic based reactive synthesis approach [28, 29, 37, 39], we can handle quantitative constraints such as energy limits more efficiently by utilizing an SMT-solver. Moreover, we also gain better scalability since we are using the constraint-based symbolic method to compactly represent the enormous search space of solutions.

- We built our own SyGuS solver based on the CEGIS framework. Compared to other SyGuS solvers based on the standard CEGIS framework, our solver is particularly optimized for robotic mobile manipulation problems and is able to generalize verification failures by finding similar states using domain-specific
heuristics, which greatly improves the performance of our method.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

In Chapter 2, we first formally define the Reactive Task and Motion Planning (RTMP) problems and concurrent games. Then we will explain the inputs to the RTMP problem through an example.

Chapter 3 describes in detail the new framework we developed to solve the RTMP problems, which is based on a system called Robosynth developed in our previous work [41, 42].

Chapter 4 discuss the proof rules for establishing the realizability of concurrent games.

Chapter 5 focuses on the internal of the Policy Synthesizer, which is the key component of our reactive synthesis framework. This chapter describes the algorithm used by the policy synthesizer in detail.

In Chapter 6, we discuss the experimental results for our approach. In order to evaluate the scalability of our approach, we design a set of benchmark problems with different kinds of task goals and we perform the comparison experiments with the default back-end GR(1) synthesizer of LTLMoP [43], a state-of-the-art tool for robotic reactive synthesis.

Conclusions and a discussion of future work are addressed in Chapter 7.
Chapter 2

Problem Formulation

In this thesis, we consider the Reactive Task and Motion Planning (RTMP) problem for geometric, task and motion domains involving a controllable robot operating in an environment with uncontrollable agents such as humans. This chapter formulates the RTMP problem and introduces concurrent games for modeling the RTMP problem. First, we define the RTMP problem in Section 2.1. Then, we describe the computational model we use for RTMP, concurrent games, in Section 2.2. Finally, Section 2.3 illustrates the inputs to the RTMP problem through an example.

2.1 Reactive Task and Motion Planning

We model the continuous, geometric portion of the domain using a kinematic tree or scene graph for the robot and objects in the environment. This data structure unifies our representation of the robot and its environment, simplifying the specification of domains. Nodes in the scene graph represent kinematic frames and attached geometry. Edges in the scene graph represent relative workspace poses in the special Euclidean group \(SE(3)\). Geometry attached to scene graph nodes represents physical objects.

**Definition 2.1.1** (Scene Graph). \(\Theta = (Q, N, F)\), where

- \(Q\) is a space of configurations
- \(N\) is a finite set of frame labels that uniquely identify each frame
- \(F\) is a finite set of kinematic frames (graph nodes), such that each frame \(f_\ell = (\ell, p_\ell, s_\ell, m_\ell)\), where,
– \( \ell \in \mathcal{N} \) is the unique label of \( f_\ell \)

– \( p_\ell \in \mathcal{N} \) is the label of the parent frame of \( f_\ell \), indicating graph edge connections

– \( s_\ell : \mathcal{Q} \mapsto \mathcal{SE}(3) \), maps from the configuration space to the workspace pose of \( f_\ell \) relative to its parent \( p_\ell \), indicating graph edge values

– \( m_\ell \) is a rigid body mesh representing geometry attached to \( f_\ell \)

The absolute pose of frame \( f_\ell = (\ell, p_\ell, s_\ell, m_\ell) \) in the scene graph is defined recursively as,

\[
0_{S_\ell}(q) = \begin{cases} 
  s_\ell(q), & p_\ell = 0 \\
  0_{S_{p_\ell}}(q) \otimes s_\ell(q), & p_\ell \neq 0
\end{cases}
\]  

(2.1)

where \( 0_{S_b} \) is the pose in \( \mathcal{SE}(3) \) from \( a \) to \( b \), \( \otimes \) is the multiplication operator on elements of \( \mathcal{SE}(3) \), and frame 0 is the global or absolute frame. Then the relative pose of frame \( f_\ell \) to another frame \( f_{\ell'} \) is \( ^f_{S_\ell}(q) = (0_{S_{\ell'}}(q))^{-1} \otimes 0_{S_\ell}(q) \).

Our approach utilizes geometric information such as distance and orientation, from the relative transformation \( ^f_{S_\ell} \) between two frames \( f_\ell \) and \( f_{\ell'} \) in the scene graph, to speed up the policy synthesis approach. See Section 5.1.3 for more details.

The definition of Reactive Task and Motion Planning (RTMP) Problem is:

**Definition 2.1.2** (Reactive Task and Motion Planning Problem).

\( T = (S, E, R, P) \), where

- **Scene Description** \( S = (\Theta_s, G, L) \), where \( \Theta_s = (\mathcal{Q}_s, \mathcal{N}_s, F_s) \) is the scene graph of objects in the workspace, \( G \) is a finite set of region labels, and \( L \subseteq \ell \times G \) is a list of frame-region pairs representing a labeling of regions in the workspace.

- **Environment Model** \( E = (\Sigma_X, \theta_e, T_e) \), where \( \Sigma_X \) is the environment state space, \( \theta_e \subseteq \Sigma_X \) is the set of initial environment states, and \( T_e(x, x') : \Sigma_X \times \Sigma_X \mapsto \{\top, \bot\} \) is the environment transition relation, i.e., \( T_e(x, x') = \top \), when the transition from \( x \) to \( x' \) is valid.
• Robot Model $R = (\Theta_r, \Sigma_Y, \theta_r, A)$, where $\Theta_r = (Q_r, N_r, F_r)$ is the scene graph describing the robot, $\Sigma_Y = Q_r \times D_Y$ is the state space combining the robot configuration space and the domain of other states such as energy, $\theta_r \subseteq \Sigma_Y$ is the set of initial robot states including the robot initial configuration, and $A$ is the Action Domain defining the set of actions $a \in A : \Theta_s \times \Sigma_Y \mapsto \Theta_s \times \Sigma_Y$.

Then a task plan $t$ is a sequence of actions $a \in A$. A motion plan $m$ is a sequence of joint configurations $q \in Q_s \times Q_r$. A task and motion plan $\pi$ is a sequence of pairs $(a, m)$, where $a \in A$ is an discrete action and $m$ is the corresponding motion plan for performing action $a$.

• Task Specification $P$ is a set of desired task and motion plans that accomplish the given task. Note we use logical formulas to compactly represent the set $P$.

A system state is a pair $s = (x, y)$ of an environment state $x \in \Sigma_X$ and a robot state $y \in \Sigma_Y$.

A system trace is defined as a sequence of system states $\sigma = s_0, s_1, \ldots$ such that, $s_0 = (x_0, y_0)$ is an initial system state, i.e., $x_0 \in \theta_e$ and $y_0 \in \theta_r$, and for every state $s_i = (x_i, y_i)$, the environment states satisfy the transition relation $T_e$, i.e., $T_e(x_i, x_{i+1}) = \top$, and $\exists a_i \in A$ for the robot state $y_i$ such that $y_{i+1}$ is the robot state after performing the action $a_i$. Thus, a system trace obeys the logical constraints of the Environment Model $E = (\Sigma_X, \theta_e, T_e)$ and the Robot Model $R = (\Theta_r, \Sigma_Y, \theta_r, A)$.

A task and motion policy is a function $p$ that selects a response robot action $a \in A$ and the corresponding motion plan $m$ for every system state $s$. Then for a system trace $\sigma = s_0, s_1, \ldots$, policy $p$ generates a task and motion plan $\pi = (a_0, m_0), (a_1, m_1), \ldots$, where every pair $(a_i, m_i)$ of action $a_i$ and motion plan $m_i$ is the output of policy $p$ for system state $s_i$, i.e., $(a_i, m_i) = p(s_i)$.

A policy $p$ is a valid task and motion policy if for every system trace $\sigma$, the corresponding task and motion plan $\pi$ generated by policy $p$ is a member of the Task Specification set $P$, i.e., $\pi \in P$. 
The solution to Reactice Task and Motion Planning (RTMP) problem $T = (S, E, R, P)$ is a valid task and motion policy $p$.

2.2 Concurrent Games

We model the RTMP problem as a two-player game where the robot and the environment are competing players. Note here the two-player games are concurrent [33, 35], which means in each round, the robot and the environment choose their moves independently and simultaneously, and the moves they choose determines the successor state for the current state. As we mentioned in Section 1.2.3, compared to the turn-based games, the concurrent games cover more general and realistic cases for robotic applications.

In this work, we define a concurrent game structure in a similar way as [33, 35] did. While [35] considers more general probabilistic two-player concurrent games and [33] considers more general deterministic k-player games, here we only consider deterministic two-player concurrent games between the environment and the robot. Thus, the concurrent games considered here is a special case of the games in [33, 35]. The concurrent game is defined as follows:

**Definition 2.2.1** (Concurrent Game).

$G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \varphi)$:

- $\Sigma$ is a state space of the game.
- $\theta \subseteq \Sigma$ is the set of initial game states.
- $M_e, M_r$ are two sets whose elements represent valid moves for the environment and the robot respectively.
- $\Gamma_e, \Gamma_r$ are valid-move functions. For each state $s$, $\Gamma_e(s) \subseteq M_e$ and $\Gamma_r(s) \subseteq M_r$ represents the set of valid moves for the environment and the robot at state $s$, respectively.
• $\delta : \Sigma \times M_e \times M_r \mapsto \Sigma$ is the transition function of the game. For every state $s \in \Sigma$, $move a_e \in \Gamma_e(s)$ and $move a_r \in \Gamma_r(s)$, $\delta(s, a_e, a_r) \in \Sigma$ is the corresponding successor state.

• $\varphi$ is a set of winning plays for the robot. Note we also use logical formulas to compactly represent the set $\varphi$.

A play $\sigma$ is an infinite sequence of states: $s_0, s_1, \ldots$, such that $s_0 \in \Theta$ is an initial state and for $i \geq 0$, $s_{i+1}$ is a successor state of $s_i$ as defined by $\delta$.

A policy for the robot is a partial function: $p : \Sigma^+ \mapsto M_r$ that selects the next move for the robot for every nonempty finite sequence of states, representing the history of the game. We also require that for all sequences $\sigma \in \Sigma^*$ and states $s \in \Sigma$, $p(\sigma s) \subseteq \Gamma_r(s)$ so that the policy is compliant with the valid-move function $\Gamma_r$ for the robot.

A play $\sigma$ is compliant with policy $p$ if for all $i \geq 0$, there are a environment move $a_e \in \Gamma_e(s_i)$ and a robot move $a_r \in \Gamma_r(s_i)$ such that, the robot move $a_r = p(s_0, \ldots, s_i)$ is selected by policy $p$ and $s_{i+1}$ is the successor state $s_{i+1} = \delta(s_i, a_e, a_r)$.

A policy $p$ is a winning policy if every compliant play $\sigma$ of the policy $p$ is a winning play, i.e., $\sigma \in \varphi$.

We call $G$ is realizable if there exists a winning policy $p$ for the robot.

As shown in the above definition, the policy for the robot may depend on an unbounded amount of information about the past history of the game. However, for finite game structures, it suffices to consider finite-state policies [24], which depend only on a finite number of states in the past history of the game and can be realized by finite-state machines. In fact, as we will see in Chapter 4, for safety and liveness games, memoryless policies exist, which only depend on the current state of the game, i.e., $p(\sigma s) = p(s)$ for all $s \in \Sigma$ and all $\sigma \in S^*$. 
2.3 Example

This section will illustrate each input of the RTMP problem through the following example.

Example 2.3.1. Consider a PR2 robot operating in a kitchen environment shown in Figure 2.1. This is the kitchen scenario studied in our previous work [41], except that there are uncontrollable agents (e.g., chefs) moving around in the kitchen and the environment is no longer static. The task for the robot is to eventually pick up a cleaned dish in Dishwasher. In this task, we have three more additional requirements for the robot:

- To ensure safety, we also require that the robot always keeps a certain distance $d_{min}$ away from humans in the kitchen, e.g., the Chef shown in Figure 2.1. That is, $\text{distance}(\text{curr}_\text{rob}, \text{curr}_\text{chef}) > d_{min}$, where $\text{curr}_\text{rob}$ is the current
location of the robot and \texttt{curr_chef} is the current location of the Chef.

- The energy status of the robot \texttt{energy_state} should be always above a certain threshold $e_{\text{min}}$. That is, $\texttt{energy_state} > e_{\text{min}}$. As shown in Figure 2.1, there is a \textit{Charge Region} in the kitchen and the robot can restore its full energy status when it stays in this region.

- If the robot moves too far at each step, it takes too long for the robot to complete the motion and the environment may change considerably by the time the robot finishes the motion. It is very likely that unsafe intermediate states will appear in long-time duration motions. Thus, we require that the distance the robot travels at each step is within a certain bound $m_{\text{max}}$, i.e., $\text{distance}(\texttt{curr_rob}, \texttt{next_rob}) < m_{\text{max}}$, where \texttt{curr_rob} is the current location of the robot and \texttt{next_rob} is the next location of the robot.

For this example, the four pieces of RTMP inputs are:

**Input 1. Scene Description S:** Figure 2.1 shows the Scene Description for this kitchen environment. The blue region is the food preparation region (\textit{FoodPrep Region}) and chefs move within this region. The yellow region is the \textit{Charge Region} and the robot can recharge when in this region. \textit{Countertop}, \textit{Dishwasher} and \textit{Storage} contain stable surfaces where the objects can be placed. In our implementation, we discretize the workspace with a grid to obtain a discrete set of locations.

**Input 2. Robot Model R:** The robot description $\Theta_r$, including the collision model, the kinematic / dynamic description of the robot, is modeled as kinematic trees with rigid links. Such information is needed for solving the low-level motion planning problems for the robot.

The action domain $A$ is defined in the formal language \textit{Planning Domain Description Language} (PDDL) \cite{44}, which is the set of actions the robot can perform, such as \textit{moveTo}, \textit{pickUp} and \textit{place}, etc. PDDL is a standard specification language for
modeling classic planning problems [6]. Generally, an action carried out by the robot achieves a certain effect and many actions require a specific condition called precondition is satisfied before they can be performed. If the precondition holds before the robot performs the action, then the effect of the action is guaranteed to hold after executing the action. For example, Figure 2.2 shows the formal definition of the pickUp action. The definition of the pickUp action makes use of several constants (e.g., the location robot_hand), predicates (e.g., reaches) and functions (e.g., rob_base) that are all defined in the same action domain (See Section 3.2.1). The precondition of the pickUp action states that there is no object in the robot’s hand, the object to be picked up should not be in the robot’s hand, the robot should be able to reach the object from its current location and all other objects are not blocking the pickUp action. The effect of the pickup action is that the object is in the robot’s hand. We can define other valid actions such as moveTo, place in a similar way using PDDL.

Input 3. Environment Model E: In reactive tasks, the environment is changing over time and the robot interacts with this dynamic environment through its sensors. If the environment has unrestricted power, there is no hope that the robot can achieve
the goal of the task. Fortunately, in reality often there are some constraints on the power of the environment. The Environment Model captures the user’s high-level assumptions on the possible behaviors of the changing environment.

Although in the real case, the environment may change continuously (e.g., a person moving continuously), in order to make the problem tractable, we model the environment as a finite state system. There are multiple ways to model a finite state system. In this work, the environment model is represented as a Symbolic Transition System (STS) $E = (\Sigma_X, \theta_e, T_e)$, where $\Sigma_X$ is the environment state space that contains all possible sensor values, $\theta_e \subseteq \Sigma_X$ is the set of initial sensor values and $T_e(x, x') : \Sigma_X \times \Sigma_X \mapsto \{\top, \bot\}$ is a symbolic transition relation that represents the relation between the current sensor value $x$ and the next possible sensor value $x'$, i.e., $T_e(x, x') = \top$ when the transition from $x$ to $x'$ is valid. Both $\theta_e$ and $T_e$ can be represented as first-order formulas.

In Example 2.3.1, there are chefs moving around the FoodPrep Region. We assume the robot can sense the exact location $P$ of the chefs. Since the velocity of the chefs is limited, the distance that the chefs can move between two steps is also bounded, i.e., $\text{distance}(P, P') < d_{\text{max}}$. The Environment Model for one chef is then $E = (\{P_0, P_1, \ldots\}, \{P_0\}, \text{distance}(P, P') < d_{\text{max}})$, where $P_0$ is the initial location of the chef and $\{P_0, P_1, \ldots\}$ is the set of discrete chef locations.

**Input 4. Task Specification $P$:** We use a plan outline [41,42] to over-approximate the task specification set $P$. A plan outline is a parameterized sequence of actions with unknown parameters. For different concrete values of these unknown parameters, the plan outline represents a different concrete plan. Thus, a plan outline in fact defines a set of plans. A valid policy should determine appropriate values for these unknown parameters such that the resulting concrete plan is a member of the intersection set of the task specification $P$ and the set defined by the plan outline.

The plan outline is one of the key components in our previous work [41,42]. Basically, the plan outline allows the user to specify high-level, partial knowledge
#define d_min 10
#define e_min 5
#define m_max 5

void main(){
    Cup cleaned_cup;
    choose(cleaned_cup);
    repeat{
        Path path;
        RobotPosition curr_rob, next_rob;
        PersonState curr_chef;
        int energy_state;

        sense(curr_chef);
        get(curr_rob, energy_state);
        choose(next_rob, path);

        assert(distance(curr_rob, curr_chef) > d_min & energy_state > e_min & distance(curr_rob, next_rob) < m_max);

        if (reaches(curr_rob, cleaned_cup)){
            pickUp(cleaned_cup);
            break;
        } else{
            moveTo(next_rob, path);
            if (In(next_rob, charge_region)){
                charge();
            }
        }
    }
    @goal: Eventually holding(cleaned_cup)
}

Figure 2.3: Plan Outline for Example 2.3.1
about valid plans in the form of a simple C-like program (For more details about
the syntax and semantics of plan outlines, please refer to [42]), which prunes out a
large number of unacceptable plans. In other words, the plan outline is a syntactic,
imperative definition of a language whose members are candidate task plans.

In this work, we extended the plan outline [41, 42] with three built-in actions:
sense, get, choose. The sense and get actions are used to bind the current environment
state and robot state, respectively, to the variables declared in the plan outline. Each
choose action is transformed to a call to the policy. That is, the choose action
\textit{choose}(y) is in fact a function call \( y' := p(x, y) \), where \((x, y)\) is the current game
state and \( y' \) the next robot state. This function \( p \) is the \textit{policy} needed for the RTMP
problem.

Figure 2.3 shows a plan outline for Example 2.3.1. Line 1-3 define several constants
used in this plan outline, i.e., the minimal distance between the robot and the chef
\( d_{\text{min}} \), the minimal energy value of the robot \( e_{\text{min}} \) and the maximum distance the
robot can travel at each step \( m_{\text{max}} \). The plan outline contains the user’s high-level
knowledge of the task that in order to achieve the goal of Example 2.3.1 (Line 33,
the robot should eventually hold one cleaned cup once the task is finished), the robot
must perform the following actions:

1. Choose a cleaned cup as the target object (Line 7).

2. Then repeatedly:

(a) Bind the current state, and choose the next location and an appropriate
path (Line 14-16).

(b) If the target object is reachable (Line 21), pick it up (Line 22) and the
task is finished (Line 23). Otherwise, the robot should move to the chosen
next location (Line 25).

(c) If the robot goes to the Charge region, the robot can restore its full energy
state by performing the \textit{charge} action (Line 26-27).
What’s more, the user can specify additional requirements other than goals for the task, in the form of assertions, as shown in Line 18-19. The assertion in the above plan outline reflects the three additional requirements for the task specified in Example 2.3.1: the robot should always keep a certain distance $d_{\text{min}}$ from the chef in the kitchen ($\text{distance(\text{curr}_\text{rob}, \text{curr}_\text{chef}) > d_{\text{min}}}$), the energy state of the robot energy_state should be always above a certain threshold $e_{\text{min}}$ ($\text{energy}_\text{state} > e_{\text{min}}$) and the distance the robot travels at each step is within a certain bound $m_{\text{max}}$ ($\text{distance(\text{curr}_\text{rob}, \text{next}_\text{rob}) < m_{\text{max}}}$).
Chapter 3

Architecture Of ROBOSYNTH for Reactive Tasks

To solve the Reactive Task and Motion Planing (RTMP) Problem, we developed a new framework based on the ROBOSYNTH system presented in our previous work [41, 42] for solving Task and Motion Planning (TMP) problems. For a given TMP Problem, ROBOSYNTH will first generate a set of logical constraints that represent the search space of possible plans and then apply an SMT-solver to efficiently explore this space. However, the previous version of ROBOSYNTH is not appropriate for solving Reactive problems due to the limitation of SMT-solvers. The main difference between non-reactive and reactive problems is that, the constraints for non-reactive problems are quantifier-free first-order formulas while those generated for reactive problems are quantified first-order formulas. For quantified formulas, many first-order theories are either undecidable (e.g., Theory of Equality) or of high complexity (e.g., Theory of Linear Integers) [45]. Therefore, SMT-Solvers often take much longer time for solving quantified formulas than quantifier-free formulas and sometimes SMT-Solvers are not able to return results for quantified formulas. In order to solve the reactive problems, we developed an additional component called Policy Synthesizer for ROBOSYNTH, which is built on top of an SMT-solver [1] and can synthesize policies for reactive problems. Chapter 5 will discuss the internals of this Policy Synthesizer.

Figure 3.1 shows a block diagram representation of the updated ROBOSYNTH framework. First, Section 3.1 and Section 3.2 briefly describe the existing components (i.e., Placement Graph Generator and Constraint Generator) of the previous ROBOSYNTH system and discuss how they are connected with the new Policy Synthesizer component for reactive tasks. For more details about the previous version of
Robosynth for non-reactive tasks, please refer to [41, 42]. Then we show in Section 3.3 how to reduce a given RTMP problem to the corresponding concurrent game structure.

### 3.1 Placement Graph Generator

Given the scene description and the robot description, the placement graph generator constructs a graph called placement graph [41] that captures the motion level information of the task. The placement graph is a variant of the Manipulation Graph [46, 47], the traditional way of representing manipulation problems. The main difference between placement graphs and manipulation graphs is that placement graphs decouple the motion of the robot base from the motion of the robot arms, which greatly reduce the size of the graph. However, we must assume when the robot moves with objects in hand, the robot should maintain the canonical arm configuration shown in Figure 2.1 so that the base motion is still valid. This is because with some arm configurations, the sweeping volume of the robot will increase when the robot is grasping...
an object and thus it might cause collisions with obstacles when the robot is moving. In contrast, in the canonical configuration, the sweeping volume of the robot when the robot is grasping an object is almost the same as that when the robot is not in grasping mode.

In this thesis, we apply the placement graph generator developed in our previous work [42] to automatically construct the placement graph. That is, we treat the previously developed placement graph generator as a black box and feed it with our scene description and the robot model. The placement graph generator returns a placement graph, which is a discrete abstraction of the continuous workspace. Next we briefly describe the content of a placement graph and how it represents the motion level information. For more details about the algorithms used in the placement graph generator, please refer to [42].

There are two kinds of nodes in the placement graph: base points (b-points) and stable points (s-points). The b-points correspond to valid robot configurations with the arms maintaining the canonical configuration (See Figure 2.1). The s-points correspond to valid robot configurations that can grasp an object from a particular location of a support surface. Since the grasping configuration may vary among different kinds of objects (e.g., cups, plates, jars), each s-point is specific to a kind of objects. For simplicity, we also use an s-point to refer to the corresponding location of the object.

There are three kinds of edges in the placement graph: connection edges, access edges and blocking edges. A connection edge from a b-point $B_1$ to another b-point $B_2$ corresponds to a collision-free path from $B_1$ to $B_2$. An access edge from a b-point $B_1$ to a s-point $S_1$ indicates that $S_1$ is accessible to the robot when the robot is in $B_1$. In this case, $B_1$ is called the parent of $S_1$. A blocking edge from a s-point $S_1$ to another s-point $S_2$ indicates that the trajectory that accesses $S_1$ is blocked if there is an object placed at the corresponding location of $S_2$. Note in general, deciding these blocking relations that indicate the infeasibility of motion plans is very difficult. The
way we compute these blocking information is that we first compute the trajectories for accessing the s-points with all other objects removed and then for each pair of s-points \((s_1, s_2)\), we place an object at \(s_2\) and check whether the trajectory accessing \(s_1\) is in collision. If this is the case, we will add a blocking edge from \(s_1\) to \(s_2\) in the placement graph. Note that even there is a blocking edge from \(s_1\) to \(s_2\), it is still possible that there is a valid trajectory that can assess \(s_1\) with the presence of the object at \(s_2\) and thus this approach is incomplete. However, our experiment shows that this approach works reasonably well in our tested robotic mobile manipulation benchmarks.

As shown in Figure 2.3, the user can specify additional requirements other than task goals in the plan outline. In order to determine the robot will satisfy those constraints, we often require additional information of each path such as energy costs. This information can be represented straightforwardly by annotating edges in the placement graph. In Example 2.3.1, the connection edges in the generated placement graph are annotated with labels indicating the corresponding energy cost, and the blocking edges are annotated with labels indicating the action blocked by this edge. For other problem domains, the edges in the placement graph can be annotated with different kinds of labels.

A portion of an example placement graph for the kitchen domain is shown in Figure 3.2. \(B_1, B_2\) and \(B_3\) are b-points and \(S_1, S_2, \ldots, S_6\) are s-points. Black arrows are connection edges, dotted arrows are access edges and blue arrows are blocking edges. The labels on the connection edges indicate the energy cost of the corresponding paths. The labels on the blocking edges indicate the action blocked by the corresponding edge.

The above discussions motivate the following formal definition of placement graphs.

**Definition 3.1.1. Placement Graph:**

A placement graph is a tuple \(G = (V, E, L)\) where \(V\) is a set of nodes, \(E\) is a set of directed edges and \(L\) is a set of labels. \(V = V_B \cup V_S\) where \(V_B\) is the set of b-points
Figure 3.2: Placement Graph Example

and $V_S$ is the set of s-points. Note that $V_B \cap V_S = \emptyset$. $E = E_C \cup E_A \cup E_B$ where $E_C \subset V_B \times V_B \times L$ is the set of connection edges with labels, $E_A \subset V_B \times V_S$ is the set of assess edges and $E_B \subset V_S \times V_S \times L$ is the set of blocking edges.

3.2 Constraint Generator

There are two stages in the constraint generation procedure: preprocessing and encoding. The following two sections will describe these two stages in detail.

3.2.1 Preprocessing

As shown in Figure 2.2, the pickup action defined in the action domain $A$ makes use of several pre-defined predicates such as $\text{reaches}(b, l)$, $\text{pickup\_blocked\_by}(l, o)$. What's more, the requirements specified in the plan outline (Figure 2.3) requires the definition of the $\text{distance}$ function. The complete list of predicates and functions
<table>
<thead>
<tr>
<th>Predicate / Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reaches(b, l)</code></td>
<td>the object at location <code>l</code> is accessible to the robot at b-point <code>b</code></td>
</tr>
<tr>
<td><code>path(p, s, t)</code></td>
<td>the robot can move from b-point <code>s</code> to b-point <code>t</code> by following path <code>p</code></td>
</tr>
<tr>
<td><code>pickup_blocked_by(l₁, l₂)</code></td>
<td>the trajectory for picking up the object at location <code>l₁</code> is blocked by the object at <code>l₂</code></td>
</tr>
<tr>
<td><code>place_blocked_by(l₁, l₂)</code></td>
<td>the trajectory for placing the object to location <code>l</code> is blocked by the object at <code>l₂</code></td>
</tr>
<tr>
<td><code>In(b, r)</code></td>
<td>the robot at b-point <code>b</code> is in region <code>r</code></td>
</tr>
<tr>
<td><code>distance(r, p)</code></td>
<td>the distance between the robot at b-point <code>r</code> and the person at location <code>p</code></td>
</tr>
<tr>
<td><code>distance(r₁, r₂)</code></td>
<td>the distance between two b-points <code>r₁</code> and <code>r₂</code></td>
</tr>
<tr>
<td><code>energy_cost(p)</code></td>
<td>the energy cost of path <code>p</code></td>
</tr>
</tbody>
</table>

Table 3.1: Predicates and Functions for the Kitchen Domain

defined for the kitchen domain is shown in Table 3.1.

Those predicates and functions represent the information in the placement graph in the form of axioms. An axiom is a logical assertion for a particular predicate or function with some concrete parameters. For example, if there is an access edge from a b-point `b` to an s-point `s`, it means that the object at `s` is accessible to the robot at b-point `b` and we will add the corresponding axiom `reaches(b, s)` to the constraints.

In the preprocessing stage, the constraint generator transforms the motion level information in the placement graph \( G = (V, E, L) \) into the corresponding axioms. The complete axiom generation rules are described below.

- For every pair of b-points \( (b₁, b₂) \) in \( V_B \times V_B \), we will calculate the euclidean distance \( d \) between \( b₁ \) and \( b₂ \), and add an axiom `distance(b₁, b₂) = d` to the
constraints.

- For every pair of a b-point $b \in V_B$ and a region $r$, if $b$ is inside the region $r$, we will add an axiom $\text{In}(b, r)$ to the constraints.

- For every connection edge from a b-point $s$ to another b-point $t$ with the label indicating the energy cost $e$, we will first assign an unique path id $p$ to this edge and then add two axioms $\text{path}(p, s, t)$ and $\text{energy\_cost}(p) = e$ to the constraints.

- For every access edge from a b-point $b$ to an s-point $s$, we will add an axiom $\text{reaches}(b, s)$ to the constraints.

- For every blocking edge from an s-point $s_1$ to another s-point $s_2$ with the label indicating that the pickUp action is blocked, we will add an axiom $\text{pickup\_blocked\_by}(s_1, s_2)$ to the constraints. Similarly, for every blocking edge with the label indicating that the place action is blocked, we will add an axiom $\text{place\_blocked\_by}(s_1, s_2)$ to the constraints.

### 3.2.2 Encoding

In the encoding state, the constraint generator will reduce the given plan outline to a logical formula. Such a logical formula can be automatically generated by applying a approach called *weakest precondition predicate transformer* [45,48]. In this approach, we work backwards from the goal, finding weakest preconditions at each step, i.e., finding the largest set of states from which we can apply the next action and still reach the goal. Specifically, for a plan outline with unknown parameters, the output weakest precondition is a first-order formula with the same set of unknown parameters (free variables). Once the values of these unknown parameters are determined by our synthesized policy, we get a complete plan that accomplishes the task.
The following standard rules describe how the weakest precondition is constructed mechanically [45, 48].

- **Assignment**: To ensure the goal $g$ holds after a assignment statement $v := e$ is executed, the weakest precondition that must hold before is:
  
  $$ wp(g, v := e) = g[v \mapsto e] $$

  where $g[v \mapsto e]$ is the result of replacing all free occurrences of $v$ in $g$ with $e$.

- **Sequence of Statements**: To ensure the goal $g$ holds after a sequence of statements $S_1; S_2; \ldots, S_n$ is executed, the weakest precondition that must hold before is defined recursively as follows:
  
  $$ wp(g, S_1; S_2; \ldots, S_n) = wp(wp(g, S_n), S_1; S_2; \ldots, S_{n-1}) $$

- **If-Statement**: To ensure the goal $g$ holds after a if-statement \( \text{If } B \text{ then } S_1 \text{ else } S_2 \) is executed, the weakest precondition that must hold before is:
  
  $$ wp(g, \text{If } B \text{ then } S_1 \text{ else } S_2) = (B \rightarrow wp(g, S_1)) \land (\neg B \rightarrow wp(g, S_2)) $$

- **Assertion**: To ensure the goal $g$ holds after an assertion \( \text{assert}(B) \) is executed, the weakest precondition that must hold before is:
  
  $$ wp(g, \text{assert}(B)) = (B \land g) $$

- **Assumption**: To ensure the goal $g$ holds after an assertion \( \text{assume}(B) \) is executed, the weakest precondition that must hold before is:
  
  $$ wp(g, \text{assume}(B)) = (B \rightarrow g) $$

- **Action**: As shown in Figure 2.2, the definition of an action includes the precondition \( \text{pre} \) and the post-condition \( \text{post} \). Before executing the action, we
must assert that the precondition of the action holds. Moreover, after execut-
ing the action, we assume that the action is correctly executed and then the
post-condition holds. Therefore, each action in the plan outline is rewritten as
follows:

\[ wp(g, \text{action}(pre, post)) = wp(g, \text{assert}(pre); \text{assume}(post)) \]

Then we use the standard rules describe above to compute the weakest precon-
dition \( wp(g, \text{assert}(pre); \text{assume}(post)) \).

Those above rules are the same rules used in our previous work \cite{41, 42}. In this
work, we extended the plan outline \cite{41, 42} with three built-in actions: \textit{sense}, \textit{get},
\textit{choose}. The \textit{sense} and \textit{get} actions are used to bind the current environment state
and robot state, respectively, to the variables declared in the plan outline. Each \textit{choose}
action is transformed to a call to the \textit{policy}. That is, the \textit{choose} action \( \text{choose}(y) \)
is in fact a function call \( y' := p(x, y) \), where \( (x, y) \) is the current game state and \( y' \)
the next robot state. This function \( p \) is the \textit{policy} needed for the RTMP problem.
Since the plan outline specifies which action the robot should take at each step (see
Section 2.3), the output weakest precondition is a constraint \( T_r(y, y') \) that specifies
the transition relation of the robot.

Another thing to note is that, although there is a rule for calculating the weakest
pre-condition of a repeat loop, this rule is not completely mechanical as it requires
the programmer to supply a loop invariant \cite{45}. In our previous work \cite{41, 42}, our
solution is to unfold the repeat loop a bounded number of times and check whether
the task goal is achieved within this finite horizon. However, this approach won’t
work for the reactive case as the reactive tasks often contain an infinite number
of interactions between the environment and the robot. As a result, we need to
intelligently discover loop invariants that can be used to establish the proof for the
correctness of the generated plan. In Chapter 5, we describe the internals of another
component, \textit{Policy Synthesizer}, which automatically constructs such loop invariants
based on some syntactic templates. If the policy synthesizer successfully finds such invariants and returns the required policy for the reactive tasks, we can instantiate the plan outline with the output policy to get a concrete plan, which guarantees that the goal will be satisfied if the robot follows this plan.

### 3.3 Game Generator

The *Game Generator* apply the following steps to generate the corresponding concurrent game $G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \varphi)$ for a given RTMP problem $T = (S, E, R, P)$, where $S = (\Theta_s, G, L)$ is the *Scene Description*, $E = (\Sigma_X, \theta_e, T_e)$ is the *Environment Model*, $R = (\Theta_r, \Sigma_Y = Q_r \times D_Y, \theta_r, A)$ is the *Robot Model*, and $P$ is the *Task Specification*.

- We first discretize the continuous geometry. Our implementation uses the *placement graph* [41] data structure (See Section 3.1), which is a sampling-based approach that transforms continuous configuration space $Q_r$ to discrete abstraction $Q'_r$. The placement graph is suitable for mobile manipulation, but our overall approach applies to other discretizations which may be suitable for different domains, e.g., cell decompositions for robot navigation.

- The state space of the game is $\Sigma = \Sigma_X \times Q'_r \times D_Y$.

- The set of initial states in the game is $\theta = \{(x, y)|x \in \theta_e \text{ and } y \in \theta_r\}$.

- The transition relation $T_r(y, y')$ for the robot is constructed by the *constraint generator* (See Section 3.2) based on the given *Action Domain* $A$, the plan outline, and the edges of the placement graph which indicate valid motions of the robot.

- For every valid transition from state $s = (x, y)$ to $s' = (x', y')$, i.e., $T_e(x, x') = T_r(y, y') = \top$, we create new move symbols $a_e$ and $a_r$ for the environment and robot respectively. Then, we add these move symbols to the set of valid moves.
That is, for $\alpha \in \{e, r\}$, we add $a_{\alpha}$ to both the total move set $M_\alpha$ and the set $\Gamma_\alpha(s)$ of valid moves at state $s$. Finally, we record the moves in transition function as $\delta(s, a_e, a_r) = s'$.

- The set of winning plays $\varphi$ is task specification set $P$, i.e., $\varphi = P$.

These steps reduce the RTMP problem to the problem of determining whether the concurrent game $G$ is realizable. In next chapter, we introduce the proof rules for establishing the realizability of concurrent games with different kinds of winning conditions.
Chapter 4

Proof Rules for Concurrent games

In this thesis, we extend the proof rules of [4,5] to verify the correctness of a candidate policy. These proof rules provide a compact, efficient representation of the constraints on valid policies. Verification using the proof rules only requires reasoning over pairs of consecutive states \((s, s')\) (see Figure 4.1, Figure 4.2 and Figure 4.3) rather than all states along a play, greatly reducing the size of constraints for verification. Moreover, the premises of these proof rules are first-order formulas, so their satisfiability can be determined using SMT solvers [1]. These SMT solvers are highly efficient in practice and the underlying theories of these solvers are sufficiently expressive to reason about useful quantitative constraints in the mobile manipulation domain, such as energy limits.

The rest part of this chapter is organized as follows. Section 4.1 introduces the specification language (LTL) for winning conditions. Then Section 4.2 describes several proof rules for different kinds of games. Finally, Section 4.3 discusses our assumption on concurrent games regarding the safety of robot’s continuous executions.

4.1 Linear Temporal Logic

In this work, we choose Linear Temporal Logic (LTL) as our specification language for the requirements of the reactive tasks. There are three reasons for using LTL. First, the expressive power of LTL has been demonstrated in applications such as program repair [34], program verification [49, 50] and game solving [5]. Particularly, in the robotics community, most of the reactive synthesis approaches [28,37,39] use LTL as their specification language. Secondly, there are proof systems available for verifying
arbitrary LTL properties [4,51]. These proof systems provide several proof rules that can be used as constraints for verifying the correctness of policy functions in reactive tasks. We will explain these proof rules in detail in Section 4.2. Finally, LTL is a more compact specification language than explicit specifications such as automaton, which considerably reduces the size of requirement specifications.

In the rest part of this section, we will give a brief introduction the syntax and semantics of LTL. For more details of LTL, please refer to [52].

Let \( \Pi \) be a set of atomic propositions. A LTL formula \( \varphi \) is constructed from atomic propositions \( p \in \Pi \) as follow.

\[
\varphi ::= p \mid \neg \varphi \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \Box \varphi \mid \varphi U \varphi
\]

The Boolean operators are negation (\( \neg \)), conjunction (\( \land \)) and disjunction (\( \lor \)) and the temporal operators are next (\( \Box \)) and until (\( U \)). Furthermore, we can also derive additional temporal operators such as eventually (\( \Diamond \)): \( \Diamond \varphi = true U \varphi \) and always (\( \Box \)): \( \Box \varphi = \neg \Diamond \neg \varphi \).

A model \( \sigma \) for a LTL formula \( \varphi \) is an infinite sequence of truth assignments to the atomic propositions \( p \in \Pi \). Let \( \sigma(i) \) be the set of atomic propositions that are true at location \( i \). We write \( \sigma, i \models \varphi \) if the LTL formula \( \varphi \) holds at location \( i \). The semantics of LTL is defined inductively as follows:

\[
\begin{align*}
\sigma, i \models p & \iff p \in \sigma(i) \\
\sigma, i \models \neg \varphi & \iff \sigma, i \not\models \varphi \\
\sigma, i \models \varphi \land \psi & \iff \sigma, i \models \varphi \text{ and } \sigma, i \models \psi \\
\sigma, i \models \varphi \lor \psi & \iff \sigma, i \models \varphi \text{ or } \sigma, i \models \psi \\
\sigma, i \models \Box \varphi & \iff \sigma, i + 1 \models \varphi \\
\sigma, i \models \varphi U \psi & \iff \text{there exits } k \geq i \text{ such that } \sigma, k \models \psi \text{ and } \sigma, j \models \varphi \text{ for all } i \leq j < k
\end{align*}
\]

We say that \( \varphi \) holds on \( \sigma \) if \( \sigma, 0 \models \varphi \), denoted by \( \sigma \models \varphi \). To conclude, a LTL formula \( \varphi \) is interpreted in the same way as Propositional Logic on Boolean operators.
For the temporal operators, the formula $\Box \varphi$ means that $\varphi$ should be true in the next location, $\varphi U \psi$ means $\varphi$ should remain true until $\psi$ becomes true, $\Diamond \varphi$ means at some location of $\sigma$, $\varphi$ is true, and $\square \varphi$ means $\varphi$ should be true at every location of $\sigma$. Furthermore, the formula $\square \Diamond \varphi$ means $\varphi$ should become true infinitely many times in the sequence $\sigma$.

### 4.2 Proof rules of Different Games

LTL is one of the acceptable and frequently used specifications for program verification, especially for the verification of concurrent and reactive programs [4,51,53–56]. However, the complexity of the LTL synthesis problem is double exponential in the size of the formula [22] and this high complexity makes the LTL synthesis problem almost intractable in practice. In [26], a polynomial algorithm is presented for a restricted fragment of LTL formulas, Generalized Reactivity(1) (GR(1)) formulas. Although the complexity of the algorithm presented in [26] is polynomial in the size of the state space, in practice, the size of the state space could be exponential, which has been known as the state explosion problem of GR(1) synthesis [40].

On the other hand, there are a lot of effective attempts on verifications of LTL properties. In [4], a relatively complete proof system is presented for LTL property verification, which only requires pure assertional reasoning, not involving additional temporal reasoning. In [51], a unifying automata-theoretic framework is presented for proving temporal properties. Although the original motivation of the above works is to perform temporal verification of concurrent and reactive programs, many of the ideas behind these works can also be applied to program synthesis. A recent work [5] in constraint-based synthesis has successfully applied the proof rules presented in [4,51] to solving infinite games. Note that although we are using proof rules similar to those in [5], our back-end synthesis engine is very different because we are dealing with a very different problem domain, the mobile manipulation domain. As a result, we need to supply some kinds of domain-specific knowledge and heuristics to make the
back-end synthesizer efficient.

In this section, we present proof rules for three kinds of LTL properties: safety, liveness and general LTL properties. Those proof rules guarantee that if the premises of the rule are met, the robot has a winning policy in the corresponding game $G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \varphi)$. Each proof rule presented here is sound and (relatively) complete. We define soundness and (relative) completeness of proof rules as follows:

**Definition 4.2.1. Soundness:** A proof rule is sound if and only if the conclusion of the proof rule holds whenever the premises of the proof rule are satisfied by some auxiliary assertions.

**Definition 4.2.2. (Relative) Completeness:** A proof rule is (relatively) complete, under the assumption that the assertion language we use is sufficiently expressive, if and only if there exist auxiliary assertions satisfying the premises of the proof rule whenever the conclusion of the proof rule holds.

By utilizing these proof rules, we only need to analyze the pair of consecutive states instead of the infinite number of states along a play, which greatly reduces the size of constraints for the winning policy. Moreover, it allows us to apply the powerful SMT solvers developed recently [1] for the required assertional reasoning about those proof rules. These SMT solvers are highly efficient in practice and the underlying theories of these solvers are sufficiently expressive to reason about some interesting quantitative constraints of the mobile manipulation domain, such as energy limits, path length, etc.

### 4.2.1 Safety Property: $\square safe(s)$

For a safety property, the robot has a winning policy in the game $G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \varphi)$, where $\varphi = \square safe(s)$, if the states visited by all plays satisfy the assertion $safe(s)$. The corresponding proof rule is presented in Figure 4.1.

**Rule Safety** requires an invariant assertion $inv(s)$ that defines the set of states visited in the winning policy for the robot. The connection between the invariant
Find an assertion $inv(s)$ such that:

$S_1: \forall s. \theta(s) \rightarrow inv(s)$

$S_2: \forall s. \exists a_r. \forall a_e \exists s'. inv(s) \land (a_e \in \Gamma_e(s)) \rightarrow (a_r \in \Gamma_r(s)) \land (s' = \delta(s, a_e, a_r)) \land inv(s')$

$S_3: \forall s. inv(s) \rightarrow safe(s)$

$G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \Box safe(s))$ is realizable

Figure 4.1: Proof rule RuleSafe for a safety property $\Box safe(s)$

assertion $inv(s)$ and the reachable states is defined inductively in RuleSafe. $S_1$ is about the base case, which states that the initial states of the game should be in $inv(s)$. $S_2$ represents the inductive case, which requires that for every state in $inv(s)$, there exists a move for the robot that bring the game back to $inv(s)$, no matter what move the environment chooses. Finally, $S_3$ states the winning condition, which requires that all states visited in the winning policy satisfy $safe(s)$.

**Theorem 4.2.1.** The proof rule RuleSafe is sound and relatively complete.

**Proof.** We split the proof into two parts: soundness and completeness.

- **Soundness** We proof the soundness by contradiction. Assume that there exists an assertion $inv(s)$ that satisfies the premises of RuleSafe yet the conclusion of RuleSafe does not hold. That is, there is no winning policy for the robot and the game $G$ is unrealizable. Hence, there exists a winning policy $p_e$ for the environment such that every play that is compliant with $p_e$ eventually reaches a state that violates $safe(s)$.

We can construct a play $\sigma$ that is compliant with $p_e$ in the following way. We start from some initial state $s_0$ that satisfies the initial condition $\theta(s)$. Thus, $s_0 \models inv(s)$ due to $S_1$. Each iteration round extends the play obtained so far
by one state, say $s'$. We maintain a condition that each such $s'$ satisfies $inv(s)$. Let $s$ be the last state of the play $\sigma$ constructed so far. Due to our condition, we have $s \models inv(s)$. Then $p_e$ determines the next move $a_e \in \Gamma_e(s)$ for the environment, and S2 guarantees that there exist a move $a_r \in \Gamma_r(s)$ for the robot and a state $s'$ such that $s' = \delta(s, a_e, a_r)$ and $s' \models inv(s)$. Finally, due to S3, we have $s' \models safe(s)$.

By iteratively constructing the play $\sigma$ using the above steps, we obtain a play that is compliant with the policy $p_e$. However, we obtain a contradiction, since according to our construction all states in $\sigma$ satisfies $safe(s)$, however $p_e$ guarantees that every play that is compliant with $p_e$ eventually reaches a state that violates $safe(s)$.

• **Completeness**  Assume the conclusion of RuleSafe holds, i.e., the robot has a winning policy $p_r$. We prove the completeness claim by constructing an assertion $inv(s)$ that satisfies the premises of RuleSafe.

Consider the set $visited = \{s \mid \exists \text{ play } \sigma \text{ that is compliant with } p_r \text{ and } s \in \sigma\}$ and let $inv(s)$ be the assertion that evaluates to true at state $s$ if and only if $s \in visited$. Since $p_r$ is a winning policy for the robot, all states in $visited$ also satisfy $safe(s)$, i.e., $inv(s)$ satisfies S3. Moreover, $inv(s)$ satisfies S1, since $p_r$ guarantees that the robot wins from every initial state and thus every state satisfying $\theta(s)$ is also in the set $visited$. For an arbitrary state $s \in visited$, $p_r$ guarantees that there exists a move $a_r \in \Gamma_r(s)$ for the robot such that for every possible choice $a_e \in \Gamma_e(s)$ of the environment, we can find a successor $s' = \delta(s, a_e, a_r)$ that satisfies $safe(s)$. Moreover, $s'$ is also in the set $visited$ and thus $s' \models safe(s)$. Therefore, we conclude that $inv(s)$ satisfies the condition S2 as well.
RuleLiveness

Find assertions $inv(s)$ and $round(s, s')$ such that:

$L1: \forall s. \theta(s) \rightarrow inv(s)$

$L2: \forall s. \exists a_r. \forall a_e. \exists s'$

\begin{align*}
inv(s) \land \neg dst(s) \land (a_e \in \Gamma_e(s)) \rightarrow (a_r \in \Gamma_r(s)) \land (s' = \delta(s, a_e, a_r)) \land inv(s') \land round(s, s')
\end{align*}

$L3: \text{well-founded}(round)$

$$G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \diamond dst(s))$$ is realizable

Figure 4.2 : Proof rule RuleLiveness for a liveness property $\diamond dst(s)$

4.2.2 Liveness Property: $\diamond dst(s)$

For a liveness property, the robot has a winning policy in the game $G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \varphi)$, where $\varphi = \diamond dst(s)$, if every play eventually reaches a state that satisfies the assertion $dst(s)$. The corresponding proof rule is presented in Figure 4.2. As mentioned in [5], in order to prove such eventuality property holds, we need to use well-founded relations.

RuleLiveness requires two auxiliary assertions $inv(s)$ and $round(s, s')$. Similar to RuleSafe, $inv(s)$ is an invariant assertion that defines the set of states visited in the winning policy for the robot, and the connection between the invariant assertion $inv(s)$ and the reachable states is defined inductively, which is captured by L1 and L2. Note that L2 only imposes constraints when $dst(s)$ is not yet satisfied and thus $inv(s)$ does not contains the states reached after $dst(s)$ is met. $round(s, s')$ keeps track of the pairs of consecutive states in the winning policy, as stated in L2. Finally, to ensure that the robot eventually satisfy $dst(s)$, L3 states $round(s, s')$ is well-founded. Informally, well-foundedness of $round(s, s')$ means that there is no infinite sequence of states $\sigma = s_0, s_1, \ldots$ such that each pair of consecutive states $s_i$ and $s_{i+1}$ satisfy $round(s, s')$, for all $i \geq 0$. Thus, it is impossible to satisfy the antecedent of L2
infinitely often because \( \text{round}(s, s') \) is well-founded and \( \text{dst}(s) \) is eventually satisfied.

**Theorem 4.2.2.** The proof rule \textsc{RuleLiveness} is sound and relatively complete.

**Proof.** We split the proof into two parts: soundness and completeness.

- **Soundness** We proof the soundness by contradiction. Assume that there exist assertions \( \text{inv}(s) \) and \( \text{round}(s, s') \) that satisfy the premises of \textsc{RuleLiveness} yet the conclusion of \textsc{RuleLiveness} does not hold. That is, there is no winning policy for the robot and the game \( G \) is unrealizable. Hence, there exists a winning policy \( p_e \) for the environment such that every play that is compliant with \( p_e \) never reaches a state that satisfies \( \text{dst}(s) \).

  We can construct a play \( \sigma \) that is compliant with \( p_e \) in a similar way as done in the proof of Theorem 4.2.1. We start from some initial state \( s_0 \) that satisfies the initial condition \( \theta(s) \). Thus, \( s_0 \models \text{inv}(s) \) due to L1. Since the environment is winning, we also have \( s_0 \models \neg \text{dst}(s) \). Each iteration round extends the play obtained so far by one state, say \( s' \). We maintain a condition that each such \( s' \) satisfies \( \text{inv}(s) \) and \( \text{round}(s, s') \). Let \( s \) be the last state of the play \( \sigma \) constructed so far. Due to our condition, we have \( s \models \text{inv}(s) \). Again, since the environment is winning, we have \( s \models \neg \text{dst}(s) \). Then \( p_e \) determines the next move \( a_e \in \Gamma_e(s) \) for the environment, and L2 guarantees that there exist a move \( a_r \in \Gamma_r(s) \) for the robot and a state \( s' \) such that \( s' = \delta(s, a_e, a_r) \) and \( s' \models \text{inv}(s) \). Finally, due to L2, we also have \( (s, s') \models \text{round}(s, s') \).

  By iteratively constructing the play \( \sigma \) using the above steps we obtain a play that is compliant with the policy \( p_e \). Thus, there is an infinite sequence of states \( s_0, s_1, ... \) and each pair of consecutive states \( s_i \) and \( s_{i+1} \) satisfy \( \text{round}(s, s') \), for all \( i \geq 0 \). However, the existence of such an infinite sequence contradicts the well-foundedness condition imposed by L3.

- **Completeness** Assume the conclusion of \textsc{RuleLiveness} holds, i.e., the robot
has a winning policy \( p_r \). We prove the completeness claim by constructing assertions \( \text{inv}(s) \) and \( \text{round}(s, s') \) that satisfy the premises of RULELIVENESS.

First we construct two sets \( \text{visited} \) and \( \text{rank} \) in the following way. For every play \( \sigma = s_0, s_1, \ldots \) that is compliant with \( p_r \), let \( k \) be the position of the first occurrence of a state in \( \sigma \) that satisfies \( dst(s) \), i.e., we have \( s_k \models dst(s) \) and \( s_i \models \neg dst(s) \) for all \( 0 \leq i \leq k-1 \). Such \( k \) exits since the play \( \sigma \) satisfies \( \Downarrow dst(s) \). Then for each \( 0 \leq i \leq k \), we add the state \( s_i \) to the set \( \text{visited} \). Furthermore, for each \( 0 \leq i \leq k-1 \), we add the pair of states \( (s_i, s_{i+1}) \) to the set \( \text{rank} \). Let \( \text{inv}(s) \) be the assertion that evaluates to true at state \( s \) if and only if \( s \models \text{visited} \), and \( \text{round}(s, s') \) be the assertion that evaluates to true for a pair of states \( (s, s') \) if and only if \( (s, s') \in \text{rank} \). Note that for each pair of states \( (s, s') \) that satisfy \( \text{round}(s, s') \), we have \( s \models \neg dst(s) \).

The above construction ensures that \( \text{inv}(s) \) satisfies L1, since \( p_r \) guarantees that the robot wins from every initial state and thus every state satisfying \( \theta(s) \) is also in the set \( \text{visited} \). Now consider the state \( s \) that satisfy the left hand side of L2. \( s \in \text{visited} \) and \( p_r \) guarantees that there exists a move \( a_r \in \Gamma_r(s) \) for the robot such that for every possible choice \( a_e \in \Gamma_e(s) \) of the environment, we can find a successor \( s' = \delta(s, a_e, a_r) \) such that \( s' \in \text{visited} \) and \( (s, s') \in \text{rank} \). Therefore, the right hand side of L2 is also satisfied.

Now we show by contradiction that \( \text{round}(s, s') \) is well-founded. Assume that \( \text{round}(s, s') \) is not well-founded, i.e., there exits an infinite sequence of states \( \sigma = s_0, s_1, \ldots \) such that each pair of consecutive states \( s_i \) and \( s_{i+1} \) satisfy \( \text{round}(s, s') \), for all \( i \geq 0 \) and \( s_0 \) is an initial state. Since the set \( \text{rank} \) is constructed from valid plays, each pair of states \( (s, s') \) in \( \text{rank} \) satisfies the condition that \( s' \) is a successor of \( s \). Thus, \( \sigma \) is a valid play of the game \( G \) and this play does not visit any state that satisfies \( dst(s) \). We obtain a contradiction to the assumption that the robot has a winning policy. Hence, we conclude that L3 is also satisfied.
4.2.3 General LTL Property: \( \varphi(s) \)

In order to solve games \( G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \varphi) \) where the winning condition \( \varphi \) is a general LTL formula, we first apply a standard technique [57] for translating \( \neg \varphi \) to a Büchi automaton. Let \( B = (Q, \Sigma, init, next, acc) \) be the obtained automaton. \( Q \) is the set of states of \( B \). \( \Sigma \) is the input alphabet of \( B \) and in this case, \( \Sigma \) is also the state space of the game \( G \). Let \( q, q' \in Q \) be the states of \( B \). \( \text{init}(q) \) is the initial condition of \( B \). \( \text{next}(q, s, q') \) is the transition relation of \( B \), which transforms the state of \( B \) from \( q \) to \( q' \) while taking into consideration the current state of the game \( s \). Finally, \( \text{acc}(q) \) represents the accepting states of \( B \).

Given an infinite sequence of game states \( \sigma = s_0, s_1, ... \), which is also a valid input for the automaton \( B \), we define a run of \( B \) on \( \sigma \) to be an infinite sequence of automaton states \( q_0, q_1, ... \) such that \( q_0 \models \text{init}(q) \) and \( (q_i, s_i, q_{i+1}) \models \text{next}(q, s, q') \) for all \( i \geq 0 \). A run is accepted by \( B \) if it contains infinitely many states that satisfy \( \text{acc}(q) \). The automation \( B \) accepts a play \( \sigma \) of \( G \) if there is an accepting run on \( \sigma \). Note that since we are translating \( \neg \varphi \) to the automaton \( B \), we have if \( B \) accepts a play \( \sigma \) if and only if \( \sigma \models \neg \varphi(s) \), which means if \( \sigma \) is a winning play for the robot, then \( \sigma \) will not be accepted by \( B \).

The corresponding proof rule based on the above automata-theoretic approach [51] is presented in Figure 4.3. The basic idea behind the proof rule RuleLTL is that we construct a synchronous parallel product of the game \( G \) and the correspond automation \( B \) translated from \( \neg \varphi \). We use \( w = (s, q) \) to refer to the pair of the game variables and the states of the automaton. The premises of RuleLTL basically characterize a policy ensuring that the accepting states of \( B \) are not visited infinitely often and thus every compliant play \( \sigma \) is not accepted by \( B \), which means \( \sigma \) is a winning play for the robot.

RuleLTL requires four auxiliary assertions \( \text{inv}(w), \text{aux}(w, s'), \text{round}(w, w') \) and
RuleLTL

Find assertions $inv(w), aux(w, s'), round(w, w')$ and $fair(w, w')$ where $w = (s, q)$ such that:

B1: $\forall s, q, q'. \, \theta(s) \land init(q) \land next(q, s, q') \rightarrow inv(s, q')$

B2: $\forall w. \exists a_e. \forall a_e \exists s'$.

$inv(w) \land (a_e \in \Gamma_e(s)) \rightarrow (a_r \in \Gamma_r(s)) \land (s' = \delta(s, a_e, a_r)) \land aux(w, s')$

B3: $\forall w, w'. aux(w, s') \land next(q, s', q') \rightarrow inv(w') \land round(w, w')$

B4: $\forall w, w'. round(w, w') \land acc(q) \rightarrow fair(w, w')$

B5: $\forall w, w', w''. \, fair(w, w') \land round(w', w'') \rightarrow fair(w, w'')$

B6: well-founded($fair$)

$$G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \varphi)$$ is realizable

Figure 4.3 : Proof rule RuleLTL for a general LTL property $\varphi$

$fair(w, w')$. Similar to RuleSafe and RuleLiveness, $inv(s)$ is an invariant assertion that defines the set of game and automaton states visited in the winning policy for the robot, and the connection between the invariant assertion $inv(s)$ and the reachable states is defined inductively, which is captured by B1, B2 and B3. $aux(w, s')$ is an intermediate bookkeeping assertion that decouples the choice of next game state and the choice of next automaton state, as stated in B2. Similar to RuleLiveness, $round(w, w')$ keeps track of the pairs of consecutive game states and automaton states in the winning policy, as stated in B3. $fair(s, s')$ keeps track of the play segments that visits the Büchi accepting states, as stated in B4. Moreover, B5 ensures that once the accepting states of $B$ are visited by the current state $w$, denoted by the assertion $fair(w, w')$, $fair(w, w'')$ holds for all the following states $w''$. Finally, to ensure that the robot satisfy $\varphi(s)$, B6 states $fair(w, w')$ is well-founded. Thus, it is impossible to visit the accepting states of the translated automaton $B$ infinitely often and $B$ will not accept the play, which means $\varphi(s)$ is satisfied.
Theorem 4.2.3. The proof rule RuleLTL is sound and relatively complete.

Proof. We split the proof into two parts: soundness and completeness.

- **Soundness** We proof the soundness by contradiction. Assume that there exist assertions $inv(w)$, $aux(w, s')$, $round(w, w')$ and $fair(w, w')$ where $w = (s, q)$ that satisfy the premises of RuleLTL yet the conclusion of RuleLTL does not hold. That is, there is no winning policy for the robot and the game $G$ is unrealizable. Hence, there exists a winning policy $p_e$ for the environment such that every play that is compliant with $p_e$ violates the winning condition $\varphi$.

We can construct a play $\sigma$ that is compliant with $p_e$ in a similar way as done in the proof of Theorem 4.2.1. Since $\sigma \models \neg \varphi$, there exists an accepting run of $B$ on $\sigma$. Since $B$ is non-deterministic, for each state $s \in \sigma$ and $q$ is the current automaton state, there is a set of appropriate automaton states that every element $q'$ in that set satisfies $next(q, s, q')$. Consider each choice leads to a tree construction, as described below.

We start from some initial state $s_0$ that satisfies the initial condition $\theta(s)$. For each $q_1$ such that there exists $q_0$ such that both $init(q_0)$ and $next(q_0, s_0, q_1)$ evaluate to true, we add $(s_0, q_1)$ as a root of our tree. We also remember that $q_0$ was used to create $(s_0, q_1)$. Then, $(s_0, q_1) \models inv(w)$ due to B1. Each iteration round extends the tree obtained so far by one level. Let $(s', q')$ be a node of the current level. We maintain a condition that each such $(s', q')$ satisfies $inv(w)$ and $round(w, w')$. Let $(s, q)$ be a node of the last level constructed so far. Due to our condition, we have $(s, q) \models inv(w)$. Then $p_e$ determines the next move $a_e \in \Gamma_e(s)$ for the environment, and B2 guarantees that there exist a move $a_r \in \Gamma_r(s)$ for the robot and a state $s'$ such that $s' = \delta(s, a_e, a_r)$ and $((s, q), s') \models aux(w, s')$. Then for each $q'$ such that $next(q, s', q')$ holds we add $(s', q')$ as a child node of the corresponding node $(s, q)$. Due to B3, we have both $inv(s', q')$ and $round(((s, q), (s', q')))$ hold.
By iteratively constructing the trees using the above expansion steps we obtain a set of trees where each branch is an infinite sequence \((s_0, q_1), (s_1, q_2), \ldots\) that comes with the corresponding initial automaton state \(q_0\). Moreover, every branch corresponds to the same play \(\sigma\) that is compliant with \(p_e\) and \(\sigma \models \neg \varphi\). Thus, there exists a branch for which the infinite sequence of automaton states \(q_0, q_1, \ldots\) is an accepting run of \(B\) on \(\sigma\). Therefore, there are infinitely many states of this run that satisfy \(acc(q)\). Let \((s_{i_0-1}, q_{i_0}), (s_{i_1-1}, q_{i_1}), \ldots\) such that \(q_{i_j} \models acc(q)\) for each \(j \geq 0\). Now consider each consecutive pair of nodes \((s_{i_j-1}, q_{i_j})\) and \((s_{i_j+1-1}, q_{i_j+1})\). Since \(q_{i_j} \models acc(q)\), we have \((\langle s_{i_j-1}, q_{i_j} \rangle, \langle s_{i_j}, q_{i_j+1} \rangle) \models \text{fair}(w, w')\) due to B4. For each \((s_{k-1}, q_k)\) such that \(i_j < k < i_{j+1}\), we have \((\langle s_{k-1}, q_k \rangle, \langle s_k, q_{k+1} \rangle) \models \text{round}(w, w')\) due to our construction condition. Then by following B5, we can conclude that \((\langle s_{i_j-1}, q_{i_j} \rangle, \langle s_{i_{j+1}-1}, q_{i_{j+1}} \rangle) \models \text{fair}(w, w')\) by induction. Thus, we find an infinite sequence \((s_{i_0-1}, q_{i_0}), (s_{i_1-1}, q_{i_1}), \ldots\) such that every consecutive of nodes \((s_{i_{j-1}}, q_{i_{j}})\) and \((s_{i_{j+1}-1}, q_{i_{j+1}})\) satisfy \(\text{fair}(w, w')\). However, the existence of such an infinite sequence contradicts the well-foundedness condition imposed by B6.

- **Completeness** Assume the conclusion of RuleLTL holds, i.e., the robot has a winning policy \(p_r\). We prove the completeness claim by constructing assertions \(inv(w), aux(w, s'), round(w, w')\) and \(\text{fair}(w, w')\) where \(w = (s, q)\) that satisfy the premises of RuleLTL.

First we construct four sets \(visited, aux\_set, round\_set\) and \(rank\) in the following way. Consider each play \(\sigma\) that is compliant with \(p_r\). Since \(p_r\) is the winning policy for the robot, we have \(\sigma \models \varphi\) and thus \(\sigma\) is not accepted by \(B\). Hence, either there is an infinite run \(q_0, q_1, \ldots\) that is not accepted by \(B\), or there is a finite run \(q_0, q_1, \ldots, q_n\) that can not be extended, i.e., there is no automaton state \(q_{n+1}\) such that \(\text{next}(q_n, s_n, q_{n+1})\) holds. In either case, for each \(i \geq 0\) (and \(i \leq n\) if the run is finite), we add \((s_i, q_{i+1})\) to the set \(visited\), add \((\langle s_i, q_{i+1} \rangle, s_{i+1})\) to the set \(aux\_set\), add \((\langle s_i, q_{i+1} \rangle, \langle s_{i+1}, q_{i+2} \rangle)\) to the set \(round\_set\). Then we define \(rank\)
for the obtained \textit{round\_set} as the least solution of B4 and B5. We also require that \textit{fair}((s_0, q_1), w) holds for every \( w \neq (s_0, q_1) \). Let \textit{inv}(w) be the assertion that evaluates to true at state \( s \) if and only if \( s \in \textit{visited} \), \textit{aux}(w, s') be the assertion that evaluates to true for \((s, q, s')\) if and only if \((s, q, s') \in \textit{aux\_set}\), \textit{round}(w, w') be the assertion that evaluates to true for \((s, q, s')\) if and only if \((s, q, s', q')\) and \textit{fair}(w, w') be the assertion that evaluates to true for \((s, q, s', q')\) if and only if \((s, q, s', q') \in \textit{rank}\).

The above construction ensures that \textit{inv}(w) satisfies B1, since \( p_r \) guarantees that the robot wins from every initial state and thus every state satisfying \( \theta(s) \) is also in the set \( \textit{visited} \). Now consider the state \( s \) that satisfy the left hand side of B2. \( s \in \textit{visited} \) and \( p_r \) guarantees that there exists a move \( a_r \in \Gamma_r(s) \) for the robot such that for every possible choice \( a_e \in \Gamma_e(s) \) of the environment, we can find a successor \( s' = \delta(s, a_e, a_r) \) such that \( s' \in \textit{aux\_set} \). Therefore, the right hand side of B2 is also satisfied. Due to our construction, we also have \textit{inv}(w), \textit{aux}(w, s'), \textit{round}(w, w') and \textit{fair} satisfies B3, B4, and B5.

Now we show by contradiction that \textit{fair}(w, w') is well-founded. Assume that \textit{fair}(w, w') is not well-founded, i.e., there exists an infinite sequence \((s_0, q_1), (s_1, q_2), \ldots\) such that each pair of consecutive states \( w_i \) and \( w_{i+1} \) satisfy \textit{fair}(w, w'), for all \( i \geq 0 \) and \( w_0 = (s_0, q_1) \) where \( s_0 \) is an initial state of game and there exists \( q_0 \) such that both \textit{init}(q_0) and \textit{next}(q_0, s_0, q_1) evaluate to true. Since the set \textit{rank} is constructed from valid plays, each pair of states \((s, q), (s', q')\) in \textit{rank} satisfies the condition that \( s' \) is a successor of \( s \) and \textit{next}(q, s', q') holds. Thus, we can construct a valid play \( \sigma = s_0, s_1, \ldots \) of the game \( G \) that is compliant with \( p_r \) and a valid run \( q_0, q_1, \ldots \) of \( B \) from the sequence \((s_0, q_1), (s_1, q_2), \ldots\) and this run \( q_0, q_1 \) is accepted by \( B \) due to B4. Therefore, \( \sigma \models \neg \varphi \) and we obtain a contradiction to the assumption that the robot has a winning policy \( p_r \). Hence, we conclude that B6 is also satisfied.
Assertion Construction

We use the proof rules presented in this chapter as the constraints for verifying a candidate policy for the robot. Note that each proof rule relies on some auxiliary assertions such as $inv$, $round$, etc. Therefore, we also need to construct these auxiliary assertions when applying these proof rules as constraints for policy verification.

For safety and liveness games, we could use the safety requirement $safe(s)$ as our initial candidate for the auxiliary assertion $inv(s)$, since S3 states that $\forall s. inv(s) \rightarrow safe(s)$. Then we can iteratively update $inv(s)$ whenever we found a state that all available moves at this state can not achieve the goal (exclude this state from $inv(s)$).

For liveness games, the well-founded relation in problems with a distance metric can be constructed using the distance $distanceToGoal(s)$ between the current state and the goal region. Since we consider bounded workspace, it is impossible to have an infinite sequence of states where the value of $distanceToGoal(s)$ keeps decreasing, which guarantees the well-foundedness imposed by L3. More generally, we could add an additional integer valuable $k$ to the game state that indicates after $k$ steps, the distance to the goal region must decrease. The transition rule for $k$ is that if the next state’s $distanceToGoal$ decreases, $k' = k_{\text{max}}$ where $k_{\text{max}}$ is the maximum number of steps that the robot is allowed to deviate from the goal region. Otherwise, $k' = k - 1$. Then the template for the ranking relation used in RuleLiveness is $\text{round}(s, s') = (distanceToGoal(s') < distanceToGoal(s)) \lor ((k > 0) \land (k' < k))$. For simplicity, we assume that $k_{\text{max}}$ is big enough such that there is a valid policy that satisfies the premises of RuleLiveness with the above $round$ assertion template. In general, we could incrementally increase $k_{\text{max}}$ until we find a valid policy.

For general LTL games, the syntactic templates for the auxiliary assertions used in RuleLTL are obviously domain-specific and require the user to design those appropriate syntactic templates.
4.3 Assumption On Concurrent Games

Note that the winning conditions of concurrent games only specify the admissible (infinite) sequences of discretized game states. However, for robotics applications, we also need to consider the safety constraints in the continuous physical world. For instance, in the kitchen domain of Example 2.3.1, we could specify in the winning condition that the robot should not collide with the chef at each discretized state. More importantly, we also want to guarantee that there is no collision between two consecutive states, which is beyond the expressive power of the concurrent game structure.

For instance, in Figure 4.4, the black point indicates the robot’s location and the blue point indicates the chef’s location. Game state $s_i$ and $s_{i+1}$ are two consecutive states in a play that satisfies the safety constraints, i.e., the robot should not collide with the person at each state. Thus, there is no collision in state $s_i$ and $s_{i+1}$. However, the safety constraints of this concurrent game do not guarantee there is no collision during the continuous execution from $s_i$ to $s_{i+1}$.

Therefore, we must assume that the safety constraints in the winning condition should be strong enough to guarantee the safety of continuous executions between two consecutive states. Consider the kitchen domain of Example 2.3.1 again. Suppose the maximum distance that the chef can travel between two states is $c_m$. If we require that the distance between the robot and the chef should be greater than a certain value $d$ at each state, i.e., $distance(\text{robot}, \text{chef}) > d$ where $d > c_m$. Then we can guarantee that there is no collision between two consecutive states.
Figure 4.4: Safety of Continuous Executions
Chapter 5

Policy Synthesizer

In reactive tasks, the biggest challenge is the adversarial environment, i.e., in the worst case, the environment’s behaviors will prevent the robot from achieving its objectives. In order to model this uncontrolled environment, for each given reactive task, we need to figure out conservative but reasonable assumptions about how the environment evolves over time. Since the state of the environment changes over time, the robot’s behavior should depend on both the current state of the environment and the current state of the robot itself. Thus, for a given reactive task, the solution plan is no longer a linear sequence of actions that the robot needs to follow. Instead, the solution is a policy which takes the current state of the environment and the current state of the robot as input, and outputs the next state of the robot.

This chapter addresses the challenge of synthesizing policy functions that can guarantee that the robot will satisfy the goals and other constraints for all possible environment behaviors that meet the assumptions of the environment. In this thesis, we reduce this synthesis problem to finding a winning policy for the robot in a two-player game.

The representation of policies varies in different approaches. For instance, in [28] policies are represented as automaton and in [5] policies are represented as skolem relations. In our case, policies are represented as functions and we require the user to provide a syntactic template for the policy functions. In fact, we are solving a syntax-guided synthesis (SyGuS) [3] problem defined in the robotics mobile manipulation domain. Note that in most cases, the reactive synthesis problems are modeled as games and the different representations of policies mentioned above are just dif-
ferent implementations of the winning policies for the games. Therefore, there is no fundamental difference between our policy functions and other policy representations. Note that in this work, the policy functions are represented symbolically, which greatly reduces the size of the policy representations.

The proof rules mentioned in the previous chapter can be used as the constraints for constructing the winning policy for the robot. For instance, given a safety concurrent game $G = (\Sigma, \theta, M_e, M_r, \Gamma_e, \Gamma_r, \delta, \square_{safe}(s))$, we instantiate the proof rule RuleSafe in Figure 4.1. If the premises of RuleSafe are satisfied, there exists a winning policy for the robot. The premises S1 and S3 are universally quantified formula over the game states, while S2 contains nested quantifiers. We can eliminate the existential quantifiers via skolemization [58] as follows to get a new clause $S2'$:

$$S2 : \forall s. \forall a_e. inv(s) \land (a_e \in \Gamma_e(s)) \rightarrow inv(\delta(s, a_e, p(s)))$$

where the existentially quantified variable $a_e$ is replaced the skolem function $p$, which also defines the policy for the robot. Note that another existentially quantified variable $s'$ is determined by the transition function $\delta$ and thus we replace $s'$ in $S2$ with $\delta(s, a_e, p(s))$. Now the concurrent game synthesis problem is reduced to the following satisfiability problem: find a function $p$ and the auxiliary assertions (e.g., $inv(v)$) used in the proof rule such that the new premises of RuleSafe S1, S2' and S3 are satisfied.

This is indeed a classical program synthesis problem. In general, the program synthesis problems are extremely difficult because of the enormous search space for the program candidates. However, lots of recent work [5, 59–61] on program synthesis shows that allowing the user to provide a syntactic template for the desired program, greatly reduce the search space of valid implementations and thus makes many difficult program synthesis programs tractable in practice. This synthesis approach is called syntax-guided synthesis (SyGuS) [3]. Motivated by these works, we model the concurrent game synthesis problem as a SyGuS problem and use syntactic templates to constrain the search space of the desired winning policy $p$ and auxiliary assertions.
These syntactic templates are represented in the form of context-free grammars.

Although there are many developed solvers [30–32] available for solving the SyGuS problem, all these solvers are developed for synthesizing general programs and are not capable of synthesizing programs in the robotic domain. This is because in the robotic domain, we also need to consider the geometric constraints in the continuous physical space, which greatly increases the complexity of the synthesis problem. In fact, we did try the three prototype implementations of SyGuS solvers and none of these solvers are able to return a winning policy even for small size problems. We believe that in order to efficiently synthesize the winning policy for the robot, we need to apply some domain-specific knowledge such as the geometry information learned from the low-level motion planner, to reduce the size of the search space. Thus, we build our own SyGuS solver that is particularly optimized for some robotics domains and is able to take advantage of the domain-specific knowledge learned from the motion level. The following section will describe this custom SyGuS solver (Policy Synthesizer) in detail.

5.1 Implementation Overview

Our solver is based on the Counterexample-Guided Inductive Synthesis (CEGIS) [27] framework. The CEGIS framework, developed in [27], is a very popular approach to inductive synthesis. Many synthesis approaches mentioned above [5,30] are based on this CEGIS framework. Perhaps the most powerful part of the CEGIS framework is its ability to learn from counterexamples returned by the verifier and thus in most cases, it converges very quickly.

The core steps of our policy synthesis algorithm are shown in Figure 5.1. The solver iterates through candidate generation, proof verification, and generalization until finding a valid policy or a proof that no such policy exists. This approach is based on the method of Counterexample-Guided Inductive Synthesis (CEGIS) [27]. We extend the conventional CEGIS approach with our generalization step, using
domain-specific heuristics which give order-of-magnitude performance improvements in our benchmark domain (Section 6.2.5).

5.1.1 Candidate Generation

We represent the policy candidate using conditional expressions that choose different outputs based on the current state values. Thus, the grammar for the policy is:

\[ p(s) \rightarrow \Gamma_r(s)[i] \]

\[ \mid \text{if } \text{test}(s) \text{ then } p(s) \text{ else } p(s) \]

where \( \Gamma_r(s)[i] \) is the \( i \)th valid move of the robot at state \( s \), i.e., the \( i \)th element of move set \( \Gamma_r(s) \), and test\((s)\) is a first-order formula that defines a set of game states. Thus, a policy \( p(s) \) expands either to a terminal symbol \( \Gamma_r(s)[i] \) or recursively to a conditional expression on \( s \).

Our candidate generator in Figure 5.1 adapts the enumeration algorithm presented in [30] to generate candidate policies. At each iteration, the candidate generator systematically enumerates candidate policies that are consistent with the generated counterexamples, in increasing order of expression size. If no such candidate exists, it reports that the game is unrealizable. Otherwise, it passes the candidate policy \( p \) to the Proof Verifier.
Note that for some problem domains, it is possible to apply heuristics to generate “good” candidate strategies. Suppose we have a heuristic cost function \( c : \Sigma \times M_r \mapsto N \) that assigns each available move \( a_r \in \Gamma_r(s) \) at the state \( s \) a integer cost \( c(s, a_r) \). This cost \( c(s, a_r) \) represents our estimation of the cost to achieve the goal at state \( s \) if we choose the move \( a_r \). For example, if the goal is to eventually reach a particular region, the distance to the goal region is a natural choice of a heuristic to help us generate good candidate policies.

### 5.1.2 Proof Verification

The next step is to check whether the candidate policy \( p \) is correct. The proof verifier in Figure 5.1 instantiates constraints constructed from the corresponding proof rule (see Section 4.2) with the candidate policy \( p \). If the constraints are valid against all admissible environment behaviors, we have found a valid policy for the robot. Otherwise, the verifier generates a counterexample: a game state \( s \) that violates the constraints.

### 5.1.3 Generalization

Now, we generalize the returned counterexample \( s \) to a set of similar examples using domain-specific heuristics. This generalization step is a major difference between our Policy Synthesizer and the conventional CEGIS framework. In the conventional CEGIS framework, after a counterexample is generated, only this counterexample is added to the example set for guiding the search of the policy. In contrast, our generalization of \( s \) to multiple similar counterexamples reduce the necessary number of iterations and improves the scalability of our synthesis algorithm.

To see how the generalization procedure works, consider a pursuit-evasion game (an instance of safety games) where we synthesize a winning policy for the evader. The verifier discovers that (see Figure 5.2), if the pursuer is at location \( l_p \) and the evader is at location \( l_e \), the output of the candidate policy \( p(l_p, l_e) \) is incorrect (i.e.,
the evader will be captured by the pursuer). Then for other state \((l_{p'}, l_{e'})\) that has the same relationship, i.e., distance\((l_{p'}, l_{e'}) = d\) and orientation\((l_{p'}, l_{e'}) = o\) (Note these geometric values such as distance and orientation are extracted from the scene graph, see Definition 2.1.1), the output of the candidate policy \(p(l_{p'}, l_{e'})\) is also incorrect. This is because the policy is represented symbolically and thus the states that have the same relationship between the pursuer state and the evader state are assigned the same next move. Therefore, we also need to include such \((l_{p'}, l_{e'})\) into our example set. Note that this generalization procedure requires domain-specific heuristics. For different problem domains, the generalization heuristics can be different.

5.2 Algorithm Discussion

Our Policy Synthesizer is sound and complete, meaning that the synthesizer will terminate with a winning policy if the game is realizable, or a proof that no such policy exists if the game is unrealizable, under the following assumptions:

- The state space of the game \(\Sigma\) is finite. In the worst case, our synthesizer needs to check every game state.

- The search space of winning policies defined by the provided grammar contains valid policies.

- The discrete abstraction – in our implementation, the placement graph – captures the continuous motions for a winning policy.
For finite games, this algorithm will terminate because, in the worst case, the verifier will return every state $s \in \Sigma$ as a counterexample. Then the example set already contains all game state and the check performed in the CandidateGenerator will either confirm that the candidate policy $p$ is correct or report that there is no solution. The key insight here is that it is impossible for the verifier to return a state $s$ as a counterexample more than once. This is because each time the verifier is called, the candidate $p$ should pass the check in the CandidateGenerator and thus is correct for all states in the example set $I$. Thus, the state in $I$ can not falsify the specification $P$ instantiated using this candidate policy $p$.

As shown in Section 4.2, as specifications, these proof rules are sound and complete. Thus, if we find a policy $p$ such that the specification is valid, $p$ is indeed the winning policy for the robot. However, since we utilize the syntactic template for reducing the search space of auxiliary assertions, for general LTL games, it is possible that the syntactic templates may not contain the witness assertions for the winning policy. Therefore, in general, this approach is incomplete. This is a usual trade-off between completeness and efficiency when dealing with difficult problems such as program synthesis. It is straightforward to implement a naive approach that enumerate all possible candidates for those auxiliary assertions and it is complete. However, the performance of this naive approach will be very poor for the real world problems.
Chapter 6

Experimental Results

This chapter presents the experimental results of our approach. In order to evaluate the scalability of our approach, we designed a set of benchmark problems with different kinds of task goals and we performed comparison experiments with the back-end of LTL MissiOn Planner (LTLMoP), a state-of-the-art synthesis tool for robotic applications [43]. Additionally, to demonstrate the ability of our new ROBOSYNTH framework to handle quantitative constraints, we performed simulations on the kitchen environment described in Figure 2.1 with additional requirements about energy limits.

We first briefly describe the implementation of our framework and experimental setup for the above tests.

6.1 Implementation and Experimental Setup

As mentioned in Chapter 3, our framework for reactive tasks is based on a system developed in the previous work [41] called ROBOSYNTH. In order to handle reactive tasks, we developed an additional component called Policy Synthesizer for ROBOSYNTH, which is built on top of the SMT-solvers and can synthesize policies for reactive problems. This policy synthesizer was implemented in OCaml [62] and utilize Z3 4.4 [1] as the SMT-solver.

Our framework was tested in the kitchen environment shown in Figure 2.1 with a model of the PR2 robot. In this kitchen environment, there are other uncontrollable agents (e.g., the chefs) moving around within the Food Preparation Region. The robot must avoid collisions with these people. We abstract the continuous movement
of the robot and these people by a set of discretized locations. This set of locations is obtained by making a grid of the workspace and we assume that the moving speeds of the robot and these people are limited so that both the robot and these people can only move to the grids adjacent to the current locations at each round.

We test our framework for three kinds of tasks. The first kind only requires safety goals: the robot should never collide with other people in the kitchen. The second kind requires that the robot satisfies both the above safety goals and one additional liveness goal: the robot should eventually pick up a target object. The third kind requires the robot satisfy both the safety constraints and one additional GR(1) property: the robot should visit several marked regions infinitely often. More details about these three kinds of tasks will be described in the next section.

Moreover, in order to demonstrate the ability of our new Robosynth framework to handle quantitative constraints, we performed simulations on the kitchen environment described in Figure 2.1 with additional requirements about energy limits. That is, we require that the robot’s energy status should be always greater than a certain threshold value. When the robot’s energy status is close to this threshold, the robot should recharge in the Charge region shown in Figure 2.1.

We compared our framework with the back-end of LTL MissiOn Planner (LTL-MoP), a state-of-the-art synthesis tool for robotic applications [43]. LTLMoP [43]. LTL MissiOn Planner (LTLMoP) is an open-source toolbox for the design, synthesis and implementation of high-level robot controller w.r.t tasks specified as GR(1) formulas. The front-end of LTLMoP is written in python while the default back-end GR(1) synthesizer is written in Java and is implemented on top of the JTLV [63] framework. The algorithm [26] used in the back-end synthesizer is polynomial-time in terms of the state space size. However, in practice, the size of the state space could be exponential, which has been known as the state explosion problem of GR(1) synthesis [40]. We will compare the performance of our Policy Synthesizer with the back-end synthesizer of LTLMoP. Note that our framework and LTLMoP are imple-
mented in different languages. Therefore, when comparing these two approaches, we should focus on the scalability of both methods rather than the absolute performance difference between these two approaches.

We note also that LTLMoP contains many front-end features, such as natural language processing, that are orthogonal or even complementary to the policy synthesis work we present in this thesis.

All experiments were carried out on a 8 core 3.4 GHz machine with 8 GB of memory. Next section will discuss the detailed results of all these experiments. For all the benchmarks, the size of the workspace is fixed (64 grids), and there are 2 chefs in the kitchen. We tested both our Policy Synthesizer and the back-end GR(1) Synthesizer of LTLMoP with increasing size of the Food Preparation Region. As the size of the Food Preparation Region increases, the size of the state space also increases and the synthesis problem becomes more difficult.

6.2 Results

6.2.1 Benchmark: Safety Property

In the safety property benchmark, there are two task requirements for the robot:

- Always avoid collisions with the people.
- When the current region is dirty, clean the current region.

The performance results for the safety property benchmark are shown in Figure 6.1. Note that the result graph is plotted on a semi-log scale. For small size problems (FoodPrep Region size $\leq 16$), the GR(1) Synthesizer performs better than our Policy Synthesizer. This is because for these small size problems, the state space of these problems is very small. GR(1) synthesizer runs very fast on problems with a small state space. In contrast, there is some overhead in our framework since we need to do a lot of preprocessing work such as encoding before we actually run our
synthesizer. For small size problems, these preprocessing steps use a large percentage of the synthesis time (about 50%).

However, for problems where FoodPrep Region size is greater than 16, our method performs better. Moreover, for problems with size above 32, the GR(1) Synthesizer took more than 10 minutes while our Policy Synthesizer solves all problems within 10 seconds. This is because for these relatively large size problems, the state spaces of these problems blow up very quickly and LTLMoP suffers a lot for this state explosion problem [40]. For instance, we can see a dramatic increase in the synthesis time when the number person states increases from 32 to 36 in Figure 6.1. However, since we are using a constraint-based symbolic method to compactly represent such large state space, our framework scales better for these large size problems.
6.2.2 Benchmark: Liveness Property

In the liveness property benchmark, there are two task requirements for the robot:

- Always avoid collisions with the people.
- Eventually pick up a cleaned dish from the dishwasher.

![Figure 6.2: Performance Results for the Liveness Property Benchmark](image)

The performance results for the liveness property benchmark are shown in Figure 6.2. This result graph is also plotted on a semi-log scale. The scalability results of this benchmark are similar to the results of the safety benchmark. For small size problems, the GR(1) synthesizer performs slightly better than our Policy Synthesizer while for problems where FoodPrep Region size is greater than 16, our method starts to perform better.
6.2.3 Benchmark: GR(1) Property

In the GR(1) property benchmark, there are two task requirements for the robot:

- Always avoid collisions with the people.
- Visit the marked regions infinitely often.

The performance results for the GR(1) property benchmark are shown in Figure 6.3. This result graph is also plotted on a semi-log scale. The scalability results of this benchmark are similar to the results of the above two benchmarks. For small size problems, the GR(1) synthesizer performs slightly better than our Policy Synthesizer while for problems where Food Prep Region size is greater than 16, our method starts to perform better.
6.2.4 Benchmark: Liveness Property with Quantitative Constraints

In order to demonstrate the scalability of our method to RTMP problems with quantitative constraints, we run the liveness property benchmark with one additional quantitative constraints on the energy state of the robot:

- The robot’s energy state $energy\_state$ should be always greater than a certain threshold value $e_{\text{min}}$. That is, $energy\_state > e_{\text{min}}$ (quantitative constraints).

As shown in Figure 2.1, there is a Charge region in the kitchen where the robot can recharge to restore its full-energy state.

The performance results for the liveness property benchmark with energy constraint are shown in Figure 6.4. For comparison, we also include the results for the liveness property benchmark without the energy constraint. As we can see from Figure 6.4, with the additional energy constraint, our policy synthesizer still scales well. Compared to the performance on the benchmark without the energy constraint, the performance with energy constraint is roughly one-time slower.

6.2.5 Generalization

In order to demonstrate the gains from the generalization step (i.e., using domain-specific heuristics to generalize verification failures), we test our Policy Synthesizer with generalization turned on off on the liveness property benchmark.

Figure 6.5 shows the performance results of our Policy Synthesizer without generalization. For comparison, we also include the results of our method with generalization. Note that the result graph is plotted on a semi-log scale. The results demonstrate that generalization step gives order-of-magnitude performance improvements and without generalization, our method can not solve the problem with FoodPrep Region larger than 12 in 30 minutes.
6.2.6 Simulation Evaluation

To demonstrate the ability of our new ROBOSYNTH framework to handle linear constraints, we performed simulations on the kitchen environment described in Figure 2.1 with additional requirements about energy status. For these simulations, the task requirements for the robot are:

- The robot should avoid collisions with the people in the kitchen environment.
- The robot should eventually pick up a cleaned dish from the dishwasher.
- The robot’s energy status $robot\_energy$ should be always greater than a certain threshold value $e_{\text{min}}$. That is, $robot\_energy > e_{\text{min}}$ (linear constraints). As shown in Figure 2.1, there is a Charge region in the kitchen where the robot can get charged and restore the full-energy status when it stays at this region.
We performed two simulations for this task. In one case, initially the robot’s battery is charged with enough energy so the robot can accomplish the above task without visiting the Charge region. In the other case, the robot’s initial energy status is relatively low. Thus, if the robot does not visit the Charge region and get charged, before it reaches the dishwasher, at some point the energy status of the robot will be below the threshold, violating the third requirement of this task.

Figure 6.6 shows the execution of this task for the first case, where the robot is initially equipped with enough energy. The initial state of the task is illustrated in Figure 6.6a and Figure 6.6b shows an intermediate state of the task. Since initially the robot’s battery is charged with enough energy, the robot moves directly to the dishwasher (Figure 6.6c) and grasps a cleaned dish in the dishwasher (Figure 6.6d).

The execution of this task for the second case, where the robot initially does not
Figure 6.6: Simulation: Enough Initial Energy

have enough energy for accomplishing the task, is illustrated in Figure 6.7. The initial state of the task is shown in Figure 6.7a. Since the robot’s initial energy status is relatively low, the policy synthesized by our framework decides to visit the Charge region and get charged first (Figure 6.7b). Then the robot moves to the dishwasher
Figure 6.7 : Simulation: Visiting the Charge Region

(Figure 6.7c) and grasps a cleaned dish in the dishwasher (Figure 6.7d).

The above two simulations show that our framework is able to handle linear constraints such as energy status requirements. In these simulations, the number of robot states is 28 and the number of person states is 6 (there is only 1 person in the envi-
ronment). The synthesis time for these two simulations is about 1.8 seconds, which is acceptable in practice.
Chapter 7

Conclusions and Future Work

In this thesis, we have presented a constraint-based symbolic approach to address the Reactive Task and Motion Planning (RTMP) problems where the robot needs to accomplish tasks under dynamic environments. The key idea behind our approach is that we model the RTMP problem as a two-player game between the robot and the environment and thus the problem of constructing a valid policy for RTMP is reduced to the problem of finding a winning policy for the robot in this two-player game.

Determining whether the robot has a winning policy is not easy, especially for the concurrent games we considered in this thesis. In order to determine the realizability of the game (i.e., the robot is winning), we have adapted the standard proof rules [4,51] for Temporal Property Verification to solve the current games. The premises of these proof rules are used as the constraints that restrict the search space of winning policies.

As shown in Section 4.2, these proof rules are sound, i.e., the robot has a winning policy if the premises of the proof rule are satisfied by some auxiliary assertions, and relatively complete under the assumption that the assertion language we use is sufficiently expressive, i.e., assume all the auxiliary assertions needed in the proof rule can be expressed using the assertion language we choose, then there exist auxiliary assertions satisfying the premises of the proof rule if the robot has a winning policy.

Given these constraints, the problem is to find a winning policy such that the constraints hold no matter what the environment behaves. This is indeed a classical program synthesis problem and it can be modeled as a syntax-guided synthesis (SyGuS) problem. We have developed our own SyGuS solver based on the popular
Counterexample-Guided Inductive Synthesis (CEGIS) framework [27]. This solver is particularly optimized for robotic mobile manipulation problems and is able to take advantage of the domain-specific knowledge learned from the motion level to speed up the synthesis process.

Our results show that, compared to an existing robotic synthesis tool – the back-end GR(1) Synthesizer of LTLMoP – our approach scales better for the benchmark problems. We show also that our approach can handle quantitative constraints such as energy limits efficiently.

For future work, first we plan to add the support for lazily constructing the placement graph. In the current version of ROBOSYNTH, the placement graph is pre-computed and since we only spend a limited amount of time for computing this placement graph, it is possible that some feasible motions are not included in this placement graph. If all valid task plans require some of these feasible motions excluded from the placement graph, ROBOSYNTH will report that there is no solution. Thus, the current version of ROBOSYNTH is complete only up to the plans in the placement graph, which offers a relatively weak guarantee. One recent ongoing work in our group shows that if we could compute the motions on demand and iteratively increase both task planning steps and motion planning time, we are able to guarantee probabilistic completeness. We plan to pursue this promising direction and try to integrate ROBOSYNTH for reactive tasks with this probabilistically complete framework.

In this work, we assume that the robot has perfect sensing. For instance, in Example 2.3.1, we assume that every time the robot performs a sense action, it can get the exact location of the chef and there is no uncertainty in this sensing information. However, in practice, it is often the case that there is noise in the sensor output. How to extend the current implementation of ROBOSYNTH to handle uncertainty in the environment, which includes not only sensor noise but also other kinds of probabilistic beliefs that we assume or observe, still remains a question.
Investigating this direction may make ROBOSYNTH capable of solving more realistic problems since there are many uncertainties in the real world.
Bibliography


