ROBOT CONTROLLERS: ONLINE AND OFFLINE ADAPTION, AND AUTOMATIC CODE TRANSFER

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Kai Weng Wong

August 2018

ROBOT CONTROLLERS: ONLINE AND OFFLINE ADAPTION, AND AUTOMATIC CODE TRANSFER

Kai Weng Wong, Ph.D.

Cornell University 2018

The development of robotic technology has evolved and changed how robots operate next to humans. In the past decades, robots were mainly seen in factories with confined and structured environments. Increasingly, robots are moving out of situated environments and are starting to operate in unstructured and dynamic environments. With the change of operating environment, many more environment events can arise during robot task execution. Robots need to cope with these events, sometimes anticipated and sometimes not, and be able to finish their assigned tasks. In this work, I present three approaches to increase robustness in robot task execution when unexpected situations arise. The approaches span low-level robot controllers and high-level robot controllers, and include both offline and online solutions.

The first contribution is an offline approach that automatically adapts and transfers robot programs between robots, leveraging the Robot Operating System (ROS). The approach reduces the time spent on retrofitting existing programs on new robots and speeds up the time from robot software development to execution.

The robot programs considered in the first contribution are often referred to as low-level robot controllers, retrieving sensor information from the robot and sending commands to the robot. They are usually executed alongside with high-level controllers, which process commands or specifications from users. The low-level and high-level controllers are

inter-connected but their relationships and interactions are rarely inspected.

The second contribution is an approach that inspects low-level controllers and proposes changes to the corresponding high-level specification, based on potential conflicts among the low-level controllers. The approach addresses the disconnect between high-level and low-level controllers which can lead to execution errors.

The final contribution is an approach that increases robustness when unexpected environment events arise in the execution of high-level robot controllers. These events may involve uncontrolled environment behaviors or other robots operating in the same workspace, resulting in unpredictable robot behaviors during task execution. The approach automatically adapts the robot controller when these environment events occur, such that the robot can finish its task safely. Throughout the work, the approaches demonstrate how to cope with different unexpected situations and increase robustness during robot task execution.

BIOGRAPHICAL SKETCH

Kai Weng (Catherine) Wong was born in Macau, Macau in 1991. She grew up there and completed her primary and secondary schooling in Chan Sui Ki Perpetual Help College in Macau, before attending De Anza College in Cupertino, California from 2008 to 2010. She transferred to Cornell University and earned her Bachelor of Science in Mechanical Engineering in 2013. She continued her graduate study at the Sibley School of Mechanical and Aerospace engineering at Cornell and received her doctoral degree in 2018.

For my parents and Flora,

Jim and FriedEgg, and

so many other people who have inspired me along the way.

ACKNOWLEDGEMENTS

First, I thank my family for their continuous support during my doctoral study. My graduate study came as a bit of surprise and thanks for bearing with me. I also want to thank Jim. Thank you for reading my papers even when you didn't want to and being there unconditionally. All your support has made this possible.

For my Cornell community, I thank my advisor Hadas Kress-Gazit. I am not the best writer or even close to be one. Thank you for bearing with all my grammatical mistakes and awkward writing, and giving me suggestion both academically and in life. I thank my committee members Joe Halpern and Ross Knepper who gave me constructive feedback and guidance during my doctoral study. Other people who meant a lot to me are my group mates in the Verifiable Robotics Research Group. Thank you all, particularly Scott Hamill and Adam Pacheck. I always had last minute paper reading requests, but you all would read my papers without hesitation. I thank my mentors Brad Treat, Eric Young, Ken Rother, Tom Schryver and Steve Gal for their guidance and support during my Commercialization Fellowship. You have showed me how live and create impact in the world. The last person that I would not miss is Marcia Sawyer. Marcia, thank you for making sure I am on track for the past 5 years.

I want to express my gratitude for my collaborators: Ankur Mehta, Rüdiger Ehlers and Hila Peleg as well. Ankur, I enjoyed our collaboration and I wish we could work together again. Rüdiger, you have inspired me and showed me how a good researcher should be. Hila, I enjoyed our time working together and how we came from different research communities and worked together seamlessly.

Last but not least, I thank my volleyball mates. You have turned a stressful PhD life into

a lively and exciting one. I think I should thank myself too. Thanks for holding on to it.

This work was supported by the NSF Expeditions in Computing project ExCAPE CCF-1138996; DARPA N66001-12-1-4250 and NSF CAREER award CNS-0953365.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

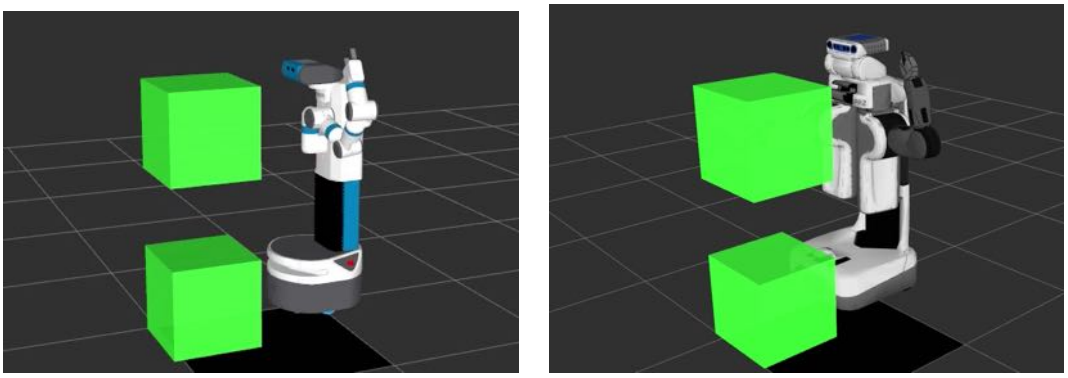
CHAPTER 1

INTRODUCTION

The development of robotic technology has not only reduced human exposure to dangerous and hazardous scenarios but also increased accuracy and efficiency of task execution. With a variety of tasks that can leverage robotic technology, humans have built many different types of robots. Some of the robots are similar to each other while others are drastically different in capabilities and appearance. For example, stationary robotic arms and mobile robots have different capabilities, while robotic arms have similar capabilities but they may vary in size. Even though robot development has improved robot hardware reliability and software stability, execution failure can still arise when robots conduct different tasks or are in proximity of other robots. Unexpected events can occur before and during robot execution. If robots do not react accordingly, this can lead to catastrophic failure and destroy both the robots and its operating environment. In this dissertation, I explore ways to create robust task execution with robots over a range of different scenarios.

First, my dissertation presents an approach to automatically convert existing source robot programs to target robot programs in Chapter 2. This is based on my current submission to the IEEE Robotics and Automation Letters (RA-L) in 2018 [92] with my collaborator Hila Peleg and my advisor Hadas Kress-Gazit. When a source robot originally designated for a task is unavailable, it is desirable to execute the source program on another target robot easily and automatically. The target robot should maintain its safety requirements and finish the task with the same or similar program. For example, as shown in Fig. 1.1, both a Fetch Robotics' Fetch robot and a Willow Garage's PR2 robot have mobile bases and are capable

of object manipulation with their end-effectors. If a Fetch robot is no longer available, it is ideal to take a readily-available program, adapt the program and execute it on a PR2 robot. An automatic transfer of programs between the robots can reduce reprogramming time and the delay on task execution.



(a) Fetch's arm away from obstacles **(b)** PR2's arm away from obstacles

Figure 1.1: MoveIt! Trajectory Planning from a Fetch Robotics' Fetch robot to a Willow Garage's PR2 robot

The above robot programs are often referred to as low-level continuous controllers. In robot execution, these low-level controllers are usually run in parallel with one or more high-level discrete controllers, and the two types of controllers together form hybrid controllers. In hybrid controllers, the high-level controllers interpret instructions from the user and the low-level controllers communicate with the robot directly and command the robot. My dissertation considers high-level controllers synthesized from specifications written in Linear Temporal Logic (LTL), using the technique from [10]. With this technique, a controller, if successfully generated, is provably-correct, i.e., the controller execution would not contradict the specification given.

In Chapter 3, my dissertation examines the interactions between low-level and high-level

controllers. This is based on my paper published in the Journal of Software Engineering for Robotics (JOSER) in 2017 [90] with my advisor Hadas Kress-Gazit. Many researchers have proposed improvements on both low-level controller execution [2, 36, 40, 52, 65, 82] and high-level controller execution [6, 7, 34, 39, 41, 44, 45, 73, 85, 93], but the interaction between the two are rarely analyzed. If not handled properly, their interaction can create problems during execution and hinder task progress. Chapter 3 addresses this problem and outlines an approach to examine the inter-connection of high-level and low-level controllers. The approach automatically suggests changes to the high-level task specification through an analysis of the low-level controller network.

Finally in Chapter 4, my dissertation proposes a system to mitigate environment anomalies during the execution of provably-correct controllers. This is based on my paper published in the IEEE Transaction on Robotics (T-RO) in 2018 [91] with my collaborator Rüdiger Ehlers and my advisor Hadas Kress-Gazit. While we can handle some potential errors before execution as in Chapter 2 and Chapter 3, unexpected events can still happen during execution. These unexpected events can lead to a controller breakdown with no transitions to a next state, as the controller does not expect such a scenario and does not know how to proceed. Here, I am interested in unexpected environment events during the execution of provably-correct controllers. Chapter 4 illustrates how a provably-correct controller can fail gracefully when unexpected events arise. The approach allows the robot to continue executing its task and make progress towards its goals as long as the robot satisfies its safety requirements. The approach further illustrates how other robots operating in the same workspace are similar to unexpected events, and these events can be modified through communication with robots creating the events.

CHAPTER 2

AUTOMATIC ROS CODE TRANSFER BETWEEN ROBOTS

2.1 Introduction

With the continuous development of robotic technology and the decreasing cost of robot hardware, the community has evolved from building and using similar industrial robotic arms in the past thirty years to creating and using more diverse hardware. Currently, there are many different types of robots and each has similar yet different physical properties. Fig. 2.1 displays some examples of existing robots. For instance, Clearpath Robotics' Jackal and Willow Garage's Turtlebot have different appearance, but both are moving bases and can be equipped with sensors such as cameras and lidar. Similarly, Universal Robots' UR5 arm and Kinova's Jaco arm have different joint limits and link lengths, but they are both robotic arms capable of performing a variety of tasks which include gesturing and inventory fetching from shelves. Finally, the last two robots, Softbank Robotics' Nao and Pepper robots, are both humanoids but their size and mobility are dramatically different.

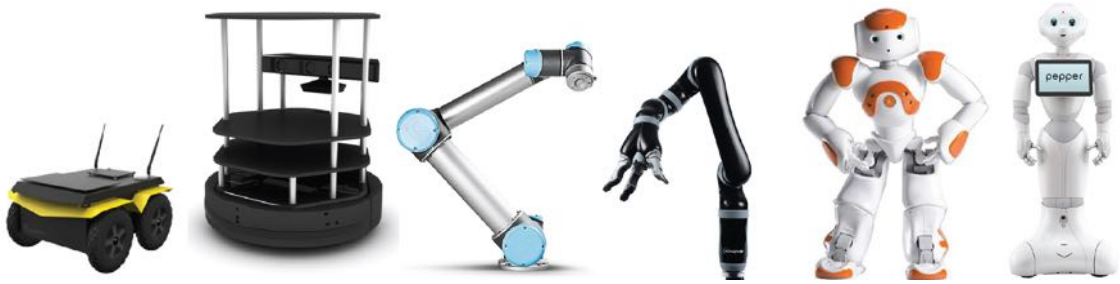


Figure 2.1: A variety of robots. From left to right: Clearpath Robotics' Jackal, Willow Garage's Turtlebot, Universal Robots' UR5, Kinova's Jaco, Softbank Robotics' Nao and Pepper.

Even though some robots have similar capabilities, transferring a program written for

one robot (source robot) to another robot (target robot) is not a trivial process since each robot typically has a different Application Programming Interface (API). To execute a source program on a target robot, one needs to rewrite the code using a different API while ensuring the original commands are valid. An easy and adaptive code transfer process between robots is desirable as it improves the robot software development paradigm and eliminates the time spent on retrofitting any existing program to another robot.

This problem is partially mitigated with the Robot Operating System (ROS) [72]. ROS is a popular open-source framework used by and developed by researchers, hobbyists and industry experts around the world. Under this framework, each robot program created follows a similar coding structure, i.e.: all robots share an API. With ROS, the transfer process is made easier but other problems still persist: a user must find and replace the right communication channels (topics/services/actions) when switching to a target robot; at the same time, the commands to the target robot from the source program can be invalid and expose the target robot to dangerous scenarios.

In this work, we present a system to streamline the code transfer process described above. We treat this problem as a *program synthesis* problem: the creation of code in a target language based on specifications. As in other program translation systems [38, 67], our specification is a program – in our case, a program written for the source robot. Given a description of the source robot, a source program and a target robot, without any source program execution, our system not only transfers code, but also considers physical properties of the source robot and the target robot. The target robot with the resulting program exhibits behaviors that are similar to the behaviors of the source robot

with the source program, with commands validated to be within the robot limits. We demonstrate how to create similar behaviors for different types of motion commands: (1) for joint commands we minimize the target robots Cartesian distance from the source robot configuration, (2) for path planning commands we ensure the target robot reaches the same goal point as the source robot, and (3) for velocity commands we scale the target robot velocities proportional to the source robot and the target robot limit ranges. In this work, we consider commands to be safe if they are within the robot hardware limits. Our system can be found on Github[1].

Related Work

Researchers have developed approaches to facilitate ROS controller setup and execution. Some researchers modified the original ROS message-passing methods to create ROS systems that tolerate online failure [51]. Other researchers streamlined the process to execute synthesized controllers from high-level task specifications with ROS [89]. Researchers also presented methods to determine ROS parameter values and internal program connections that maximize the overall system performance and minimize the required hardware resources [14]. Of all the works discussed, [14] is the closest to our work. Compared to our work, the work in [14] focuses on the ROS program resource usage and efficiency improvement, while our work focuses on the automatic transfer of ROS programs for one robot to programs for some other robot using optimization and other techniques.

Our work is closely related to some of the program translation approaches. The

[1] `https://github.com/wongkaiweng/rosCodeSyn`

translation of programs from one programming language or platform to another is a problem that has garnered recent attention [38, 67, 96], and has been tackled with machine learning tools similar to those used for natural language translation. Program repair, or the problem of accepting a program that does not satisfy some tests or logical formulas to outputting a similar program that now satisfies the tests and formulas, has also been explored in the programming languages community [54, 62, 66]. Both are fields which harness machine learning tools in order to perform program synthesis [31, 32, 50, 53, 56, 69, 70, 84, 94], generating a program that satisfies a provided specification. They deal with the general cases of their respective problems. In contrast, this paper takes a more problem-specific approach and leverages both program repair and program translation in the scope of ROS programs.

For the rest of the paper, first, we outline related preliminaries in Section 2.2. We motivate the problem solved in this paper in Section 2.3. We explain our approach in Section 2.5 with an example described in Section 2.4. Afterwards, we demonstrate our approach on several examples in Section 2.6. We evaluate our approach in Section 2.7 and discuss the challenges of code transfer in Section 2.8. Finally, we summarize our work and outline possible future extensions in Section 2.9.

2.2 Preliminaries

Definition 2.2.1. Robot Operating System (ROS) is an open-source robot control framework used by people around the world, with software packages developed by researchers and hobbyists. ROS works as a decentralized framework, with standalone programs (called

nodes) all connected to the same ROS **master** for communication purposes. In ROS, there are three ways to communicate sensor information and actuation commands between programs: topics, services and actions, which are different channels for message passing. In this work, we focus on topics and actions. In the following we describe writing a ROS program. The code snippets are all in Python.

- A **topic** is a many-in-to-many-out message channel for exchanging sensor data and robot commands. A ROS node sending messages to a channel is a channel's publisher and a ROS node retrieving messages from a channel is a channel's subscriber. A node can publish and subscribe to multiple channels. Below is a code snippet with examples of initializing a publisher and a subscriber.

```
# Publisher
rospy.Publisher(topic_name,msg_type,...)
# Publisher Example
rospy.Publisher('cmd_vel',geometry_msgs.msg.Twist,...)
# Subscriber Example
rospy.Subscriber('scan',sensor_msgs.msg.LaserScan,...)
```

In this snippet, `topic_name` is a string that specifies the topic name. `msg_type` is a class in Python and the message type. `'cmd_vel'` is an example of `topic_name` and it is of type `geometry_msgs.msg.Twist`.

- An **action** is a request and response channel with feedback provided during request processing. For this channel, a node acts as an action server while other nodes act as clients, all connected to the same ROS master. The duration from request to response varies. Below

is a code snippet with examples of initializing an action server and an action client.

```
# Action Server
actionlib.SimpleActionServer(act_name,act_type,...)
# Action Client
actionlib.SimpleActionClient(act_name,act_type,...)
# Action Client Example
actionlib.SimpleActionClient('follow_joint_traj',trajectory_msgs.msg.
   FollowJointTrajectoryAction)
```

In this snippet, `act_name` is an action name. `act_type` specifies the action type and it is a class in Python. For example, `'follow_joint_traj'` is an action of type `trajectory_msgs.msg.FollowJointTrajectoryAction`.

A special group of actions commonly used in the community is enabled by MoveIt! [80], a popular and widely-used package for trajectory planning of manipulators. In this paper, we provide insight on how to transfer programs using MoveIt! actions in addition to regular ROS actions. A typical usage of MoveIt! with Python in ROS is as follows:

```
mov_group=MoveGroupInterface(group_name,'base_link')
mov_group.moveToPose(pose_goal, pose_frame)
```

The first line above initializes a planning interface `mov_group` and specifies `group_name`, the group of joints and links involved in the planning process. The second line sends a goal pose `pose_goal` of a point (represented as a coordinate frame) on the robot, `pose_frame`, to the MoveIt! trajectory planning action server. `pose_frame` is usually a coordinate frame

on the tip of the group `group_name`.

Definition 2.2.2. ROS Messages Each message type in ROS has a defined structure [64] for storing different information. Different message types can be used together to construct a new message type. In this work, for parameter changes (Section 2.5.2), we focus on two different types of messages that are commonly used to send commands to robots in ROS: velocity messages and joint trajectory messages.

• **Velocity Messages** are in the form of Twist messages in ROS. They are sent through publishers to a robot. We can separate each Twist message into linear(v) and angular(a) velocity commands, and each of them can be further separated into x, y and z components. By convention, for a differential drive robot, a user sends linear x velocity, v_x, to move a robot forward/backward and angular z velocity, a_z, to turn a robot. Below is an example of initializing and assigning a Twist message to control a differential drive robot.

```
# Message Assignment Example
vel_msg = geometry_msgs.msg.Twist()
vel_msg.linear.x = 1.0
vel_msg.angular.z = 0.5
# Message Declaration Example
vel_msg = geometry_msgs.msg.Twist(geometry_msgs.msg.Vector3(1.0, 0, 0),
                                  geometry_msgs.msg.Vector3(0, 0 ,0.5))
```

For a holonomic robot, a user can send linear x velocity, v_x, and linear y velocity, v_y, to command a robot.

```
# Diagonal motion - holonomic robot
vel_msg.linear.x, vel_msg.linear.y = 1.0, 1.0
```

A user can also send angular z velocity (a_z) to a holonomic robot. If both linear y velocity (v_y) and angular z velocity (a_z) are sent, it is up to the internal robot software to decide on how to handle the commands.

- **Joint Trajectory Messages** specify robot joint commands and they are sent through an action server to the robot. For each message, we specify the joints to command in the field `joint_names` and the commands in the field `points`. Below is an example.

```
# Trajectory message example
traj = trajectory_msgs.msg.JointTrajectory()
traj.joint_names=['joint_1', 'joint_2', ...]
traj.points = [JointTrajectoryPoint(positions=0.0, ...),
               JointTrajectoryPoint(positions=1.57, ...), ...)
```

Each command in `points` is a `JointTrajectoryPoint` object, corresponding to a joint in `joint_names`. Note that there are other fields in `JointTrajectoryPoint` but in this work we are mainly interested in the `positions` field, or position commands. Other fields for joint velocity or acceleration commands can be handled similar to velocity commands in Section *B.2* described later.

Definition 2.2.3. Program Synthesis Problem is the problem of computing from a specification a program that implements it. The classic synthesis problem searches for an implementation to a full specification, usually encoded in some logic. Newer variations on the problem have turned to partial specifications, such as input-output examples or type information, that are easier for the user to provide but describe a much wider array of

matching programs.

One of the forms of the synthesis problem is that where the specifications are in the form of a program, as in our work here. Another work with programs as specifications is the Sketch tool [56] which is used to deobfuscate programs.

Definition 2.2.4. Abstract Syntax tree (AST) is a tree structure that represents the syntactic structure of a program or program fragment, reduced to provide the most concise and convenient form for further processing (e.g. keywords are no longer kept and parentheses are discarded). The AST can be used to further analyze the program after syntactic parsing. Additional annotations on the tree are often added to preserve semantic information found during phases of type-checking and analysis.

Definition 2.2.5. Compiler front-end is the part of the compiler that performs the analysis of the source language text, with no consideration of the target language. A phase of the front-end conducts syntactic analysis and builds an AST from the source language text. The AST and its annotations created during compilation are then used by later phases of the compiler or other analysis.

Readers can find more information about both the AST and the compiler front-end in [30].

2.3 Problem Formulation

In this work, we address the problem of automatic program adaption between robots with similar capabilities as deemed by the user. Formally,

Problem 1. Given a specification in the form of a ROS program in Python for a source robot r, automatically modify this program such that the target robot r' can safely execute the adapted program and exhibit similar behaviors as the source robot r running the original program.

In this work, we consider the target commands to be safe if they are within the robot's hardware limits. The source robot behaviors and the target robot behaviors are similar if (1) the distance between robot trajectories is minimized for joint commands, (2) the robots reach the same goal points for path planning commands and (3) velocities are scaled proportionally based on the source robot and target robot limits for velocity commands.

Our approach in this work does not require an execution of the source code on the source robot for data collection and it can handle variable assignments in the code. In this work we mainly consider ground robots but our approach is applicable to UAVS and underwater robots as well. We make the following assumptions in this work:

1. There are no inherent errors in the source code. We can compile and run the code of the source robot.

2. Depending on the commands in the source program, our system is provided with the corresponding velocity limit, the joint limit or the MoveIt! configuration files of the target robot.

Our approach currently has the following restrictions:

1. We do not deal with arithmetic calculations of variables in the code.

2. We do not modify commands generated from runtime information.

2.4 Keyboard Control Example

To illustrate our approach, consider the following ROS program taken from Github[2].

Example 1. *This program controls a Jackal robot using keyboard arrow keys. Code Snippet 1 contains parts of the program. In the code, line 1 sets up a velocity message publisher and line 3 publishes velocity messages using the publisher. In the following, we show how we automatically adapt this program and run it on a Turtlebot.*

Code Snippet 1 Keyboard Control Program

```
1 pub=rospy.Publisher('cmd_vel',Twist,queue_size=1)
2 if up_arrow_key:
3   pub.publish(Twist(Vector3(0.5, 0, 0), Vector3(0, 0 ,0)))
4 elif: ...
```

2.5 Approach

To solve Problem 1 in Section 2.3, we propose a process that synthesizes target code from source code. Our approach divides the process into two parts: First, finding and replacing the message channels with the correct names to deliver different messages. Second, ensuring the commands are valid and safe for the target robot. If possible, we modify the commands to create similar behaviors on the target robot.

[2] `https://github.com/siavash-khodadadeh/JackalNavigation/blob/master/scripts/controller.py`

2.5.1 Replace channel names based on message type (Communication)

In this subsection, we describe how we replace the subscribers/publishers and action clients/servers (channel names) of the source robot with those of the target robot. In ROS, each channel name corresponds to a message type. For example, in line 1 of Code Snippet 1, the topic name `'cmd_vel'` passes messages of type `Twist`. Our approach automatically examines and categorizes channel names by message types, and finds a suitable replacement later. We present three methods for finding channel names based on message types.

A.1 From code examples on codebases

One of the methods to find suitable channel names is to learn from other users. We can use a codebase of programs for the target robot to learn probable channel names. We created our codebase by mining online services such as GitHub and organized the code by robots. We replace the original channel name with the most likely channel name based on the probability distribution created from examples for the target robot, i.e., the channel name with the highest probability.

In this work, we have manually collected Python code from GitHub for different robots. On each collected script for the target robot, we automatically run the compiler front-end, and retrieve all channel names by message type from the generated AST. Once we collect all channel names by message types from the target robot codebase, we calculate the distribution of channel usage. In this work, we automatically select the channel name

with the highest usage probability and we replace the channel name in the source code. Alternatively, we can present the distribution to the user and prompt for user decision.

In Example 1, Code Snippet 1 is first compiled and we automatically identify all the publishers/subscribers together with action servers/clients. In this case, we find one publisher with message type `Twist`. Our method then automatically looks for all topic names with message type `Twist` from the Python codebase of the target robot, Turtlebot. We have collected 67 Turtlebot scripts and the topic name distribution of `Twist` is shown in Table 2.1.

Twist Topic Name	Count	Probability
`cmd_vel_mux/input/navi`	15	0.52
`cmd_vel_mux/input/teleop`	8	0.28
`mobile_base/commands/velocity`	3	0.10
`cmd_vel`	2	0.07
`cmd_vel_mux/input/switch`	1	0.03

Table 2.1: Distribution of type `Twist` from Turtlebot codebase

Our method automatically selects the topic name with the highest probability, replaces the topic name in the AST and converts the AST back into code for the target robot. In Example 1, our method replaces `'cmd_vel'` in line 1 of Code Snippet 1 with `'cmd_vel_mux/input/navi'`, the Turtlebot channel name with the highest count that corresponds to type `Twist`.

```
pub=rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=1)
```

A.2 From available topics/actions (rostopic)

If there aren't enough code examples to extract data, we leverage other information

available, such as the list of topics/actions connected to the current ROS master. We automatically retrieve the list of available topics/actions with the 'rostopic' command and obtain the message type of each channel. Below is an example of channel listing with the type of each channel.

```
/camera/rgb/image_raw                                   (sensor_msgs/Image)
/cmd_vel_mux/input/navi                                 (geometry_msgs/Twist)
/odom                                                   (nav_msgs/Odometry)
/scan                                                   (sensor_msgs/LaserScan)
...                                                     ...
```

Our method automatically searches for channel names matching the desired message type. Since we cannot obtain a distribution as in *A.1*, if there is more than one channel name with the same message type, we present the result to the user and ask the user to choose an appropriate replacement.

For Example 1, our method presents the following prompt to the user if we do not have any Turtlebot code examples:

```
Please select a topic for Twist from the list.
The original topic was 'cmd_vel'. Input Example: 1
1. /cmd_vel_mux/input/navi
2. /cmd_vel_mux/input/safety_controller
3. /cmd_vel_mux/input/switch
4. /cmd_vel_mux/input/teleop
5. /mobile_base/commands/velocity
```

Then our method replaces the channel name based on the user selection.

To increase replacement accuracy, we can also combine *A.1* and *A.2*: first we can use *A.1* to search for the most probable replacement and then *A.2* to verify that the replacement topic exists on the target robot.

A.3 From possible target MoveIt! groups

When the source program contains a MoveIt! trajectory planning request, we want the target robot to reach the same pose (`pose_goal`) as the source robot. The goal pose remains unchanged but to plan for the target robot, we replace the group of links and joints for planning (`group_name`), and the location of the pose goal on the robot (`pose_frame`), often the robot end effector frame, with those of the target robot. For example, for the `group_name` Nao's left arm, the `pose_frame` is Nao's left wrist.

To do so, first, we find all the possible planning groups and pose frames from the Semantic Robot Description Format (SRDF) file[3] of the target robot. An SRDF file describes all the possible planning groups and for each group, its links and joints. We compare the available target group names with the source `group_name` and sort the names by similarity using Ratcliff and Obershelp's Gestalt Pattern Matching [76]. We then present the sorted list to the user and ask for their input. We can derive a `pose_frame` automatically from the group chosen. At the end, we replace `group_name` and `pose_frame` with those of the target robot.

[3] `http://wiki.ros.org/srdf`

2.5.2 Check and modify variables (Parameters)

In addition to replacing the communication method through channel name replacement, our approach checks and ensures the commands that will be sent to the target robot are valid.

To do so, we first extract valid command ranges from different robot configuration files. We consider the Unified Robot Description Format (URDF) files[4] which encode upper and lower joint limits of manipulator joint limits, and YAML files[5] which encode velocity limits of mobile robots. We can retrieve joint limits and velocity limits from these two types of files.

In this work, we target Twist messages (velocity commands) for mobile robots and JointTrajectory messages (joint commands) for robotic arms. We outline three different methods to ensure the robot commands are within limits and to create target robot behaviors that are similar to the source robot behaviors.

B.1 Check if commands are within limits

Before modifying an existing command, first we check if it is a valid command. Our check requires the target robot's URDF file for joint command validation and its YAML file for velocity command validation. From the URDF and/or YAML file, we obtain the velocity and joint limits of the target robot. For Example 1, the YAML file excerpts of the source robot Jackal and the target robot Turtlebot are shown in Fig. 2.2.

[4] `http://wiki.ros.org/urdf`
[5] `http://www.yaml.org/`

Figure 2.2: YAML Excerpt of Jackal and Turtlebot

Jackal	**Turtlebot**

linear :
  x:
  lower: −2.0
  upper: 2.0
  deadband: 0.0
angular :
  z:
  lower: −4.0
  upper: 4.0
  deadband: 0.0
	linear :
	  x:
	  lower: −0.8
	  upper: 0.8
	  deadband: 0.0
	angular :
	  z:
	  lower: −5.4
	  upper: 5.4
	  deadband: 0.0

Similar to Section 2.5.1, we find all the commands in the compilation process. We check if a command is within the target robot limits; if it is not, we notify the user.

In Example 1, the message command on line 3 in Code Snippet 1 sets the linear x velocity component to 0.5 (`Twist.linear.x = 0.5`). We check that the command is within the velocity limits of the Turtlebot, `-0.8` $\leq$ `Twist.linear.x` $\leq$ `0.8`.

B.2 Scale velocity commands

In addition to providing feedback to the user, we can also modify and scale robot commands proportional to the source robot and the target robot limits. Scaling velocities aims to preserve relative motions in the program. This method requires YAML files of both robots.

In this work, we restrict the method to a same-drive scaling, i.e., it can be a differential-drive-to-differential-drive velocity scaling or a holonomic-robot-to-holonomic-robot velocity scaling, but it cannot be a differential-drive-to-holonomic-robot scaling. Section 2.8 outlines why it is difficult to convert commands between two different robot drives. In general, we assume both the source robot and the target robot uses the same fields, for

example `Twist.linear.x`, for robot control.

Given the robot's velocity limits and its deadband, where a command between 0 and the deadband has no effect on the robot and can be treated as zero commands, we can scale the robot commands appropriately with the following two equations:

$$C_t = \frac{(C_s - D_s)}{(UL_s - D_s)} \times (UL_t - D_t) + D_t \tag{2.1}$$

$$C_t = \frac{(C_s + D_s)}{(LL_s + D_s)} \times (LL_t + D_t) - D_t \tag{2.2}$$

In Eq. 2.1 and Eq. 2.2, we find C_t, the target command, given the source command, C_s. The subscript s stands for the source robot and the subscript t stands for the target robot. D is the deadband; UL is the upper limit and LL is the lower limit. We assume the upper limit is bigger than zero and the lower limit is smaller than zero, with the deadband being symmetric and D always positive. Eq. 2.1 scales commands bigger than or equal to zero while Eq. 2.2 scales commands smaller than zero. Our approach allows the robot to have different upper and lower limits.

In Example 1, with the source and target YAML file in Fig. 2.2, we scale the command in line 3 in Code Snippet 1 with Eq. 2.1. The new linear x velocity command is 0.2:

```
pub.publish(Twist(Vector3(0.2, 0, 0), Vector3(0, 0 ,0.0)))
```

In the previous method in *B.1*, it ensures a command is within limits but it does not guarantee the resulting behaviors to be the same: units are not encoded in the YAML files currently and the processing of commands can be different between robots. Our method in *B.2* does not rely on the source and the target YAML file using the same units and we preserve relative motions with scaling.

In this method, we outline a simple linear scaling method that scales velocity commands. Nonlinear scaling of commands is also possible based on the framework in this method. Given the limit files and the nonlinear relation between the source commands and the target commands, we can perform nonlinear conversion of commands similar to the case here.

B.3 Optimize joint trajectories

We can scale joint commands similar to velocity commands. However, straight-forward scaling of joint commands based on joint limits may result in target robot behaviors that are significantly different than the source robot behaviors. In this work, we leverage the work in [81] to map joint commands of one robot to another regardless of the number of joints and lengths of each link.

Our method requires the URDF files of both the source robot and the target robot. Before leveraging [81] and mapping joint commands, with the AST of the source code, our method automatically searches for `JointTrajectory` commands in the AST, and extracts robot joint names and joints commands to the source robot. With the source robot URDF and the list of joint names, we automatically retrieve a corresponding kinematic chain. As for the target robot, our method starts by finding source joint names from the target URDF. If we cannot find the same joints, our method automatically finds the longest kinematic chain from the target robot URDF for mapping and replacement in the transfer process.

With the kinematic chains of the source robot and the target robot, our method leverages [81] and optimizes the target joint angles, such that the resulting joint angles generate the closest robot configuration in Cartesian coordinates to that of the source robot with the source angles found in the source program. The optimization technique in [81] considers the upper and lower limits of each joint, and minimizes the distance between discretized

locations on the two chains. A user can specify the importance of the chain shape in comparison to end effector location in this optimization technique. In this work, we extend [81] and find the weight between the chain shape and the end effector automatically during optimization.

At the end, we replace both the joint names and the joint angle commands in the source code with that of the target robot from the optimization.

Table 2.2 summarizes the configuration files needed for each method described to check and modify variables. We provide feedback at the end of the transfer summarizing all the changes and replacements that cannot be found. With this information, our approach keeps users in the loop and allows the user to inspect and verify the changes before execution.

Our implementation can be found on Github[6]. We demonstrate our methods with examples in the following section.

Method	Source YAML	Target YAML	Source URDF	Target URDF
B.1 Check if commands are within limits	-	✓	-	✓
B.2 Scale velocity commands	✓	✓	-	-
B.3 Optimize joint trajectories	-	-	✓	✓

Table 2.2: Configuration files for each method

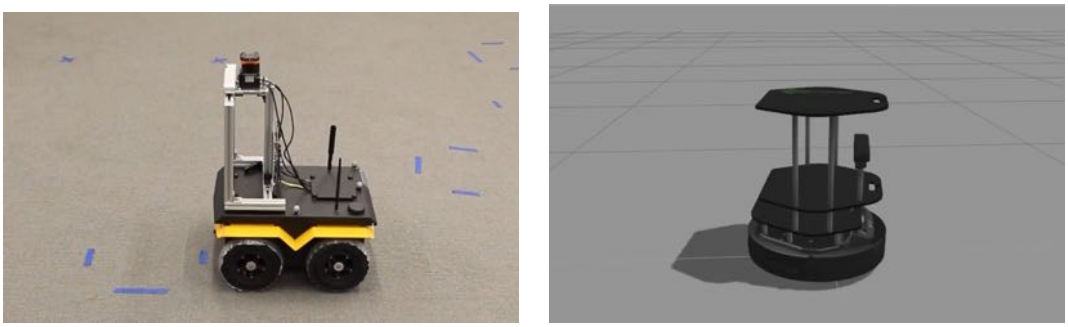
[6] `https://github.com/wongkaiweng/rosCodeSyn`

2.6 Examples

In this section, we illustrate the methods described in Section 2.5 with examples involving a variety of robots. Several source programs (Example 1-3, 5-6 below) were downloaded from GitHub, as indicated in the following. Executions of source and target robots is available online[7].

2.6.1 Keyboard control example revisited

As shown in Section 2.5, for Example 1, we can automatically replace variables and topic names of the source robot, the Jackal, with those of the target robot, the Turtlebot. After replacement, we can control Turtlebot with arrow keys and drive the robot around. This is the first example in the video online.



(a) Jackal in our lab **(b)** Simulated Turtlebot in Gazebo

Figure 2.3: Convert a Jackal program to a Turtlebot program

[7] `https://youtu.be/Bhso4Kgdcx8`

2.6.2 Obstacle avoidance example

Example 2. *We consider a ROS program that controls a Turtlebot and avoids collisions with obstacles*[8]*. We convert this Turtlebot program into a Jackal program.*

In Example 2, from the source code, we find a subscriber to the `LaserScan` topic and a publisher to the `Twist` topic. We search for suitable replacements from code examples collected for Jackal. The distributions are shown in Table 2.3 and Table 2.4.

Twist Topic Name	Count	Distribution
`cmd_vel`	4	0.57
`jackal_vel_controller/cmd_vel`	2	0.29
`cmd_vel/keyboard`	1	0.14

Table 2.3: Distribution of type `Twist` from Jackal codebase

LaserScan Topic Name	Count	Distribution
`front/scan`	1	1.0

Table 2.4: Distribution of `LaserScan` from Jackal codebase

We replace the two topics in the code with those of the highest distribution (*A.1*) and verify the topics exist on the target robot (*A.2*). We also check to ensure the commands are within limits (*B.1*) and scale velocity commands (*B.2*). The snippet below shows the modification:

```
41 rospy.Subscriber('front/scan', LaserScan, ...
...
```

[8] `https://github.com/njarrow/TurtleBot/blob/master/wander`

```
45 vel_pub = rospy.Publisher('cmd_vel', Twist, ...
...
70    vel=0.75
```

Both the source robot Turtlebot and the target robot Jackal respond to the laser scan information and avoid collision. This is the second example in the video online.

2.6.3 Joint trajectory following example

Example 3. *Consider a Universal Robot's ROS program that moves a Universal Robot UR5 arm (6 DoF) to different configurations*[9]*. We demonstrate transferring this program to a Kinova Jaco arm (7 DoF).*

To do so, we first run the compiler front-end on the source code. There is one action client with type `FollowJointTrajectoryAction` in the AST. We search through the Jaco arm codebase but we cannot find an action matching the message type (*A.1*).

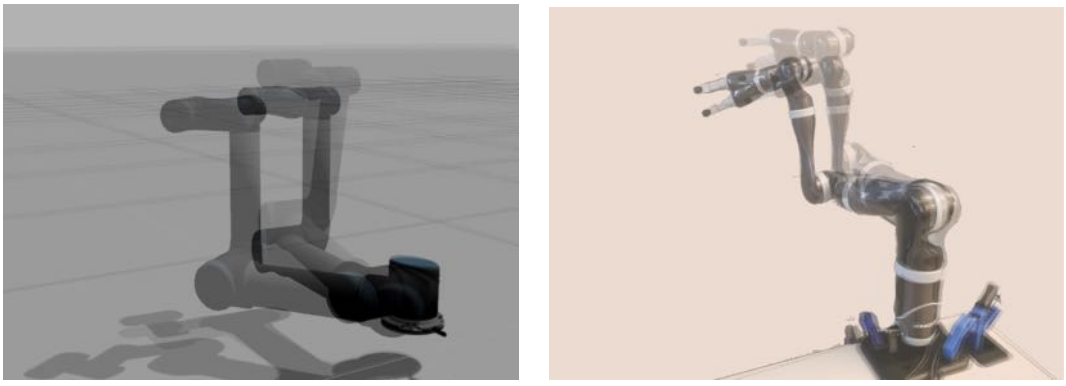
In this case, we use *A.2* and search through the current list of available topics and actions. We prompt the user to choose an action from two options found:

```
Please select an action for FollowJointTrajectoryAction from the list.
The original topic was '/arm_controller/follow_joint_trajectory'. Input Example: 1
1. effort_joint_traj_controller/follow_joint_traj
2. effort_finger_traj_controller/follow_joint_traj
```

[9] https://github.com/ros-industrial/universal_robot/blob/kinetic-devel/ur_driver/test_move.py

Since we are replacing joint commands, we choose 1 and convert the action client name from `'/arm_controller/follow_joint_trajectory'` of the UR5 to `'effort_joint_traj_controller/follow_joint _traj'` of the Jaco.

Joint names and joint commands of the UR5 are automatically extracted from the source program. We use our method in *B.3* to obtain new joint commands that create similar arm configurations as the UR5 on the target robotic arm, the Jaco. We replace the joint names and joint commands in the AST and create a new ROS program for the Jaco arm. The two robotic arms conduct a similar triangular motion at the end, as shown in Fig. 2.4. This is the third example in the video online.

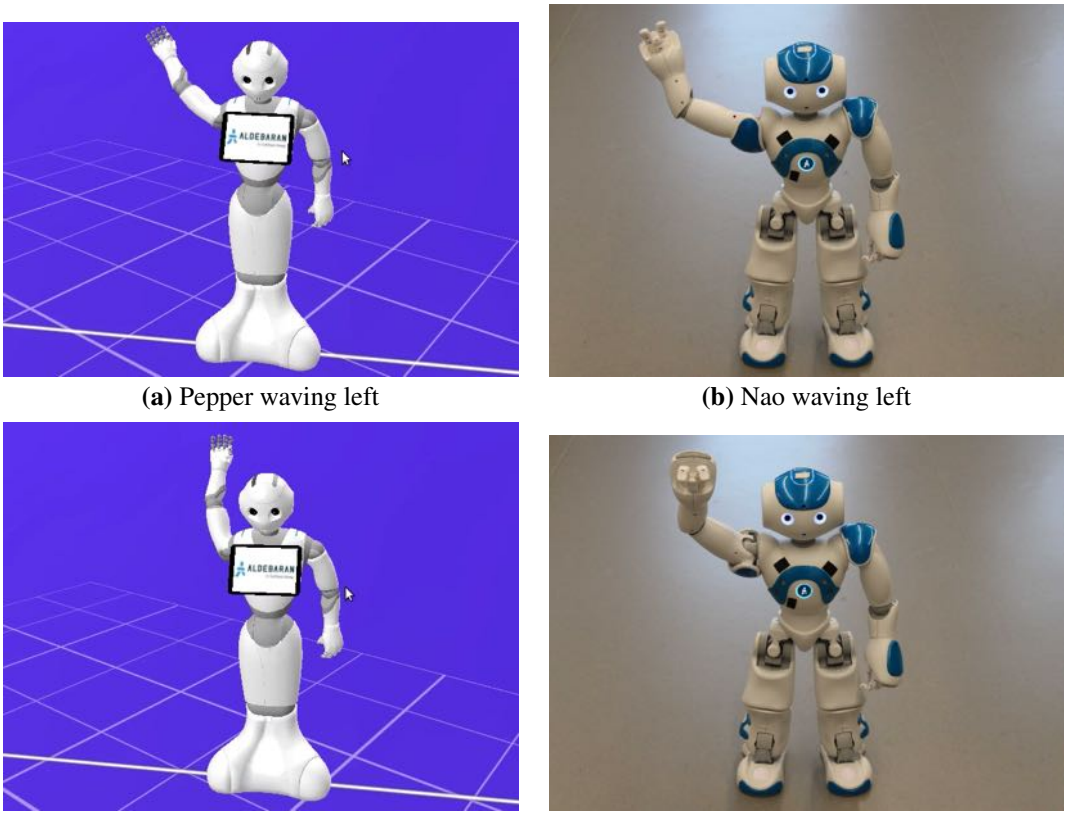


(a) Original UR5 configurations **(b)** Generated Jaco configurations

Figure 2.4: UR5 configurations to Jaco configurations with *B.3*. We removed the background of the Jaco for clarity.

2.6.4 Greet Example

Example 4. *Consider a ROS Program that controls a Softbank Robotics Pepper with a graphical interface. This is an example created by the authors. We would like to transfer*



(a) Pepper waving left

(b) Nao waving left

(c) Pepper waving right

(d) Nao waving right

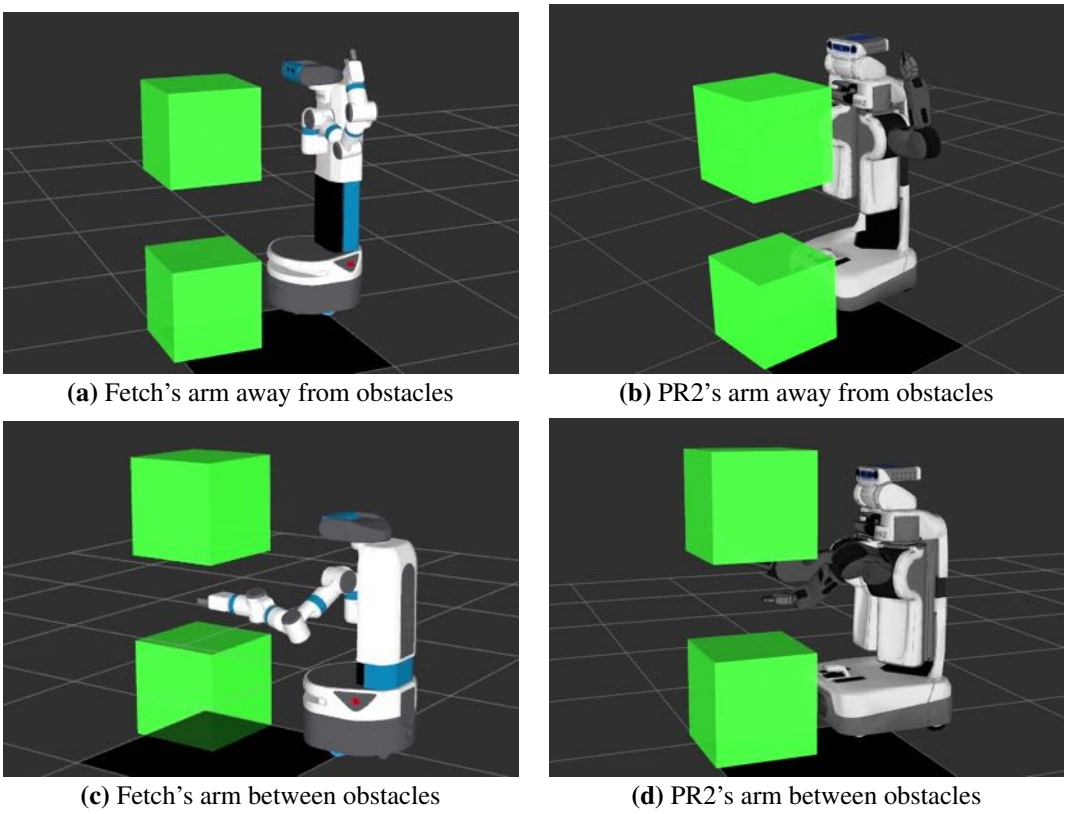
Figure 2.5: Waving motion of Pepper and Nao

this program onto a Softbank Robotics Nao.

In this example, we convert both mobile base commands in the Twist messages and manipulator commands in the JointTrajectory messages to those appropriate for the target robot. We scale velocity commands using *B.2* and perform trajectory optimization with *B.3* to create a waving motion for the Nao. The results are shown in Fig. 2.5. This is the fourth example in the video online.

2.6.5 Path Planning Example with MoveIt!

Example 5. *Consider a ROS Program that moves the end effector of a Fetch Robotics Fetch robot between two obstacles and away from them using MoveIt!. This is adapted from an example online*[10]*. We demonstrate transferring this program to a PR2 robot.*



(a) Fetch's arm away from obstacles

(b) PR2's arm away from obstacles

(c) Fetch's arm between obstacles

(d) PR2's arm between obstacles

Figure 2.6: MoveIt! Trajectory Planning from the Fetch to the PR2

Using *A.3*, we find out the code plans trajectories for the "arm_with_torso" group of the Fetch in the "wrist_roll_link" frame. Using the SRDF file of the PR2, we find all the possible

[10] `http://docs.fetchrobotics.com/manipulation.html`

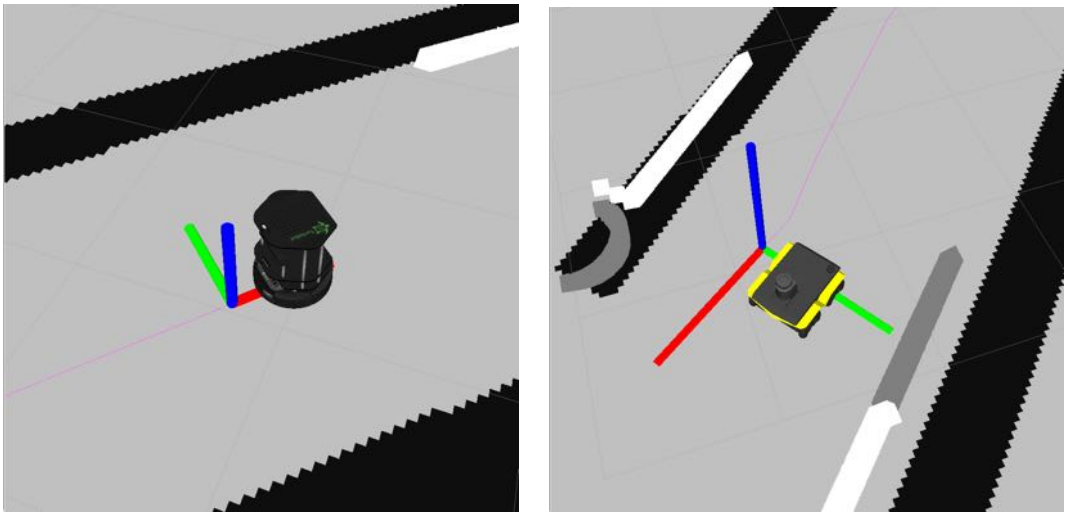
groups and we present the sorted results to the user. The user chooses "left_arm_with_torso" as the new group. From there, we automatically find a target frame "l_wrist_roll_link" and update the target robot script. The result is shown in Fig. 2.6. Note that if a goal point is not reachable, we would not know until execution. In the future, we would like to provide feedback regarding these scenarios during the transfer process. This is the fifth example in the video online.

2.6.6 Localization Example

Example 6. *Consider a ROS Program that localizes a robot based on laser scans*[11]*. The original source robot is unknown. We demonstrate transferring this program to both a Jackal robot and a Turtlebot.*

We replace the topics in the program with *A.1* and verify the topics exist on the target robot with *A.2*. We run the program and drive both the Jackal and the Turtlebot manually. During the program execution, we obtain good approximations of the robot location, as shown in Fig. 2.7. This is the sixth example in the video online.

[11] `https://github.com/penguinmenac3/ros_graph_slam/blob/master/scripts/graph_slam_node.py`



(a) Turtlebot Localization **(b)** Jackal Localization

Figure 2.7: Localization of the Jackal and the Turtlebot with laser scans. The axis indicates the robot pose estimation.

2.7 Evaluation

Table 2.7 displays a summary of the approach performance in the Example Section.

Examples	Time Taken (s)	Section 2.5.1 Communication Replacement (# of channels)	Section 2.5.2 Parameter Replacement (# of parameters)	Total Replacement (#)	User Input in Total Replacement (# of instances)	# of Target Robot Examples	Lines in File / Replacement Errors
Example 1: Keyboard control	2.9	1	8	9	0	67	75/0
Example 2: Obstacle avoidance	1.4	2	4	6	0	12	85/0
Example 3: Joint trajectory following	11.7	1	40	41	1	60	89/0
Example 4: Greet	3.3	2	14	16	1[a]	0	209/0
Example 5: Path Planning with MoveIt!	2.8	2	0	2	1[b]	110	69/0
Example 6: Localization (Turtlebot)	10.08	5	0	5	1	67	339/0
Example 6: Localization (Jackal)	8.74	5	0	5	1	12	339/0

[a]The two communication replacements both come from the Current Available Topics (*A.2*), with one been unique and required no user input.
[b]This user replacement instance comes from the MoveIt! Replacement (*A.3*), which is not based on robot examples.

Table 2.5: Summary of Example Results

As shown in Table 2.7, our approach requires minimal user intervention, with only one input from the user in Examples 3, 4, 5 and 6. In general, the need for user input is inversely proportional to the number of available target robot examples and the quality of the examples. Depending on the choice of the target robot, the user can find it difficult to collect robot examples from online codebases. Common robots such as the Turtlebot and the PR2 have relatively abundant examples available online while other robots such as the Jackal and the Nao have fewer examples by other users. Even though some robots have more examples, they are still mainly on the order of 10 with the maximum number of examples to be around 150. More examples are needed to extract other usage patterns. Collecting code examples from robotics experts also reduces replacement errors as the length of the script increases.

2.8 Challenges Ahead

To generalize a transfer process for any robot and any program, we summarize our experience and outline some of the challenges ahead:

1. *Understanding programmers' intent*

   Consider the following code snippet of Example 2:

Code Snippet 2 Obstacle Avoidance Program

```
68  ...
69  if SCAN.ranges[320]/10 >= .3:
70    vel=.3
```

In Example 2, we replaced `vel=.3` in line 70 of Code Snippet 2 with `vel=.75` for the Jackal using *B.2*. In Code Snippet 2, we can also see that there is another `.3` in line 69. We do not know the intent of the programmer here and we cannot tell if this `.3` should be replaced as well.

2. *Determining if commands are bundled*

   If we want to convert holonomic robot commands into differential drive commands, we can perform the calculation easily but the challenge comes from how we determine the commands for conversion. Consider the code snippet below:

```
vel_msg.linear.x = 1.0
...
vel_msg.linear.y = 0.5
```

In the code, the velocity message is changed at two different places. It is unclear if we should consider the two changes to be the same command when converting from holonomic commands to differential drive commands. The commands can either result in a holonomic robot moving forward then sideways or the robot moving diagonally. It may require execution of the code to figure out the actual commands and perform a conversion.

3. *Handling assignment with arithmetic operations*

As one of our assumptions, currently we do not deal with any arithmetic operations. Arithmetic operations pose extra challenge when checking if a command is within limits. They may require partial code execution to retrieve values. Another challenge arises when finding a replacement that involves multiple variables.

4. *Handling commands generated from sensor information*

Some robot commands are generated from sensor information such as lidar scans and video feed. Since we cannot get an actual command without any sensor data, it is difficult to say if a command is within limits. It may require program analysis technique such as symbolic testing or concolic testing to perform analysis and replacement.

5. *Retrieving velocity configuration in YAML files*

Currently the format for defining velocity limits is not unified and each robot manufacturer uses a different format. In this work, we have processed the files and created a unified format for using the files in our approach.

6. *Collecting robot examples*

As discussed in Section 2.7, to further improve code transfer, we need more examples than the current hundred to extract usage patterns and other information. The creation

of a centralized robot codebase would be useful for the improvement of robotic software development.

2.9 Conclusion and Future Work

In this work, we present a system that given a ROS program of a source robot, automatically synthesizes a program for a target robot. We leverage the standardized framework of ROS. Automatic code transfer improves code reusability between robots and allows fast deployment of another robot when the source robot is unavailable.

In our approach, we find and replace source robot message channels with target robot message channels. We mine possible channel names from code examples on codebases such as GitHub and current active channels. Our approach also ensures commands to the target robot are safe and the target robot exhibits similar behaviors as the source robot, by checking to make sure all commands are within limits, scaling velocities and optimizing joint trajectories.

Based on our experience, we evaluate and outline some of the challenges of automatic code transfer between robots. Our approach provides a good starting point to automatically transfer code between robots and keep the user in the loop on the transfer process. As for future work, we want to expand and find correlations among different commands with a bigger collection of code examples. We will also explore other ways to reason about transfer between different robot dynamic/kinematic models, such as from a moving base to a UAV. Last but not least, we want to collect and create a larger example codebase such that

we can extract intricate usage patterns and improve robotic software development.

CHAPTER 3

ROBOT OPERATING SYSTEM (ROS) INTROSPECTIVE IMPLEMENTATION OF HIGH-LEVEL TASK CONTROLLERS

3.1 Introduction

Synthesis of correct-by-construction robot controllers has gained popularity in recent years [3, 4, 6, 7, 23, 25, 47, 58, 59, 63, 74, 86, 93]; researchers have leveraged synthesis techniques from the formal-methods community to automatically generate these controllers. With controller synthesis, a user with no programming expertise can automatically generate a controller given a specification; if synthesis is successful, the synthesized controller is correct-by-construction, i.e., the controller does not deviate from the user instructions during execution. This is an advancement from the status quo where programmers may accidentally introduce errors into the controller, and the controller can exhibit unexpected behaviors during execution.

Over the years, researchers have explored and improved controller synthesis in a variety of directions. Controller synthesis has coupled with sampling-based motion planning to speed up motion plan creation [7] and take into account the complex and nonlinear dynamics of a system operating in a partially unknown environment [63]. Besides planning trajectories with sampling-based motion planning approaches and synthesis, we can generate trajectories by reducing the trajectory generation problem to a sequence of shorter horizon problems while maintaining temporal properties with synthesis [93]. In the case of expected occasional human intervention, we can synthesize a semi-autonomous controller

for correct operation [58]. These controllers are also improved to increase robustness against intermittent [25] or chronic [86] unexpected environment events; disturbances in a multi-agent systems with both controlled and uncontrolled agents [4]; and exogenous disturbances on a system with continuous dynamics [59]. If controller synthesis fails, we can provide feedback [74] and automatically suggest changes to the task specification [3].

A majority of existing works such as the ones above have discussed how to improve controller synthesis in different directions for robotics applications. Yet, the process to go from synthesizing a controller to executing a controller on a robot is omitted in most of the works, and this process is often not as trivial as it seems. Consider the following task that we can deploy with either a single robot or a group of robots:

Example 7. *Robot(s) always move forward. If Robot(s) sense an obstacle, it(they) should stop moving.*

Specification 1 Obstacle Sensing Specification

1 Always **move**.
2 If you are sensing **person** then do **stop**.

The high-level task specification of Example 7 is shown in Spec. 1 in the form of Structured English [46] sentences. These sentences belong to a restricted English grammar that enables users to write specifications using language and not code.

With a specification, to execute a task, we start by translating the task specification to logical formulas [46] and then synthesizing a controller using the technique from [10]. In the case of Example 7, we can successfully synthesize a controller. This controller is a

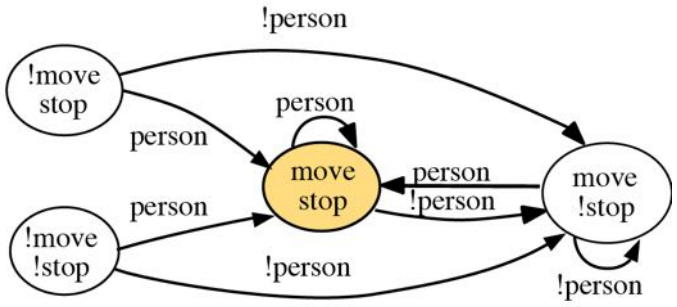


Figure 3.1: Finite state machine of Example 7. This state machine was automatically synthesized from the specification. The symbol '!' denotes that the variable value is False. The yellow node in the middle shows that with the current specification, the outputs **move** and **stop** can both be true at the same time.

finite-state machine, as shown in Fig. 3.1. To execute a continuous task with the discrete finite-state machine shown in Fig. 3.1, we abstract the continuous behaviors of the robot(s) and its(theirs) environment into Boolean variables – they are also known as 'propositions'. In the controller shown in Fig. 3.1, we have three propositions; the controller takes in as input a Boolean proposition **person** and in return determines the values of two output propositions **move** or **stop**, i.e., whether the robot should move and/or stop.

The circles in Fig. 3.1 are the robot states and there are four in total for this controller. Each circle displays the valuation of the two output propositions in that state. If a proposition's value is false, it is denoted with a '!' in front. Each edge is labeled with a valuation of the input proposition. Depending on the incoming valuation of the input proposition and the current state, the controller moves to a next state.

Consider a single-robot scenario: to execute the controller on a physical system such as a KUKA youBot, we start by connecting each input or output in the controller to an executable program that processes sensor information or sends out robot commands; we

called this process ‘mapping’. Here, we map the outputs **move** and **stop** to programs that execute robot velocity commands such as moving forward or stopping; we map the input **person** to the result of a perception module that implements a person detector using camera images from a camera mounted in front of the robot.

Even though the controller synthesized in Fig. 3.1 is correct-by-construction with respect to the specification, the user may introduce errors at runtime through the mapping of controller inputs/outputs to low-level programs executing the commands. For instance, if the outputs **move** and **stop** are mapped to the same controller, when they are both true, as in the yellow state in Fig. 3.1, conflicting velocity commands will be sent to the robot. This is undesirable and we should provide feedback to the user before execution to prevent such a case.

The synthesized controller is also not limited to single-robot execution and we can control multiple robots at the same time using a single controller such as the one in Fig. 3.1. Consider a two-robot scenario: we want to control two KUKA youbots using the controller in Fig. 3.1. In this scenario, each output proposition should command both robots. For example, if the output **move** is true, then the mapped output programs should send commands to both robots and both robots should move forward. However, the user could leave one robot uncontrolled by mistake: for example, when the output **stop** is true, commands are only sent to one of the two robots; as a result, one robot would stop while the other may keep moving forward; this generates unexpected outcomes. In this work, we also want to check for situations similar to the scenario above before execution to prevent abnormal behaviors.

In this work, we propose a framework to streamline the process of converting a high-level task specification into an execution of the corresponding synthesized controller. We focus on implementations based on the Robot Operating System (ROS), a popular open-source software with a rich library of packages developed by users around the world. ROS works as a distributed system, with programs known as 'nodes' that each must connect to a ROS master which keeps track of channels for one node to reach another node. Nodes connected to the same ROS master can communicate with each other through message passing in three different methods: topics, services and actions. A topic is a many-in-many-out long-term message channel, a service is a short request/reply channel and an action is analogous to a longer service with intermediate feedback provided before a reply.

We show in this work how there exists a natural connection of correct-by-construction controllers to ROS. Specifically, we:

1. Propose a framework for a seamless integration of correct-by-construction controllers with ROS. In this framework, we consider the controller as a ROS node, and the inputs and outputs of the controller as topics that other nodes can use to communicate with the controller.
2. Detect possible failures related to the mapping between the controller and the low-level programs, i.e., ROS nodes, connected to the controller. For example, we are able to detect possible faults related to the implementation of Example 7 where the robot can stop and move at the same time and where one robot in a group of homogeneous robots is left uncontrolled.
3. Automatically provide feedback to the user in the form of suggested changes to the

specification.

4. Demonstrate our framework for both single robot execution and homogeneous robots execution.

Compared with our previous work [89],

1. We removed the constraint of one output proposition (topic) to one ROS node. Our framework now allows connection of multiple ROS nodes to the same output proposition (topic);
2. With the constraint above removed, we also provide a solution to check if multiple nodes connected to the same output proposition are sending conflicting commands;
3. We show that our framework can control multiple robots with one correct-by-construction controller. We demonstrate the updated framework with an example controlling a group of homogeneous robots. We discuss the possible failure and feedback we can provide.

The rest of the paper is as follows: first we discuss existing work in Section 3.2; then we define the problem addressed in this paper in Section 3.3. In Section 3.4, we define the necessary preliminaries. In Section 3.5, we describe our framework and how we can detect and provide feedback to the user; we illustrate and discuss our approach using two examples in Section 3.6. In Section 3.7, we evaluate our work independent of the examples. Finally, in Section 3.8, we summarize our work.

3.2 Related Work

The ROS package SMACH [12] can execute finite-state machines; it allows a user to manually create finite-state machines and execute them on robots. Our software is similar in that we can also execute finite-state machines, but we automatically synthesize these state machines from high-level task specifications; and the controllers, if successfully generated, are correct-by-construction. The work in [87] has also executed robot controllers with ROS starting from high-level task specifications. However, they do not adapt their execution to ROS's standard structure and they do not provide any analysis.

Besides providing feedback before controller execution, some have proposed mechanisms to increase robustness in controller execution. The work in [51] modifies the ROS message-passing methods such that the modified methods can result in an adaptive system that tolerates failure online. Compare to [51], we do not modify the ROS message-passing methods; we analyze existing ROS structures and check for potential errors in this work. Researchers have also leveraged a hybrid navigation architecture for robot execution with ROS [77]. The work proposes a robot motion planning method that is reactive to obstacles. Compare to [77], our approach reacts to not only obstacles but also other environment events such as alarms or pickup requests. In this work, we propose a seamless integration of correct-by-construction controllers with ROS; we point out potential failure and suggest specification modifications before robot execution. Our changes is also not limited to motion planning.

During the execution of correct-by-construction controllers, researchers have proposed

different approaches to tackle anomalies. Researchers have monitored and detected violations of higher-level task specifications at runtime [86, 91] (Chapter 4). In their approach, they have also automatically added in 'recovery' transitions to the controller such that the robot can safely continue its task even when a higher-level task specification is violated at runtime. Some also look at resynthesizing controllers when unexpected condition occurs during execution [61]. In this work, we provide feedback and suggest changes based on the analysis of the low-level controller interactions unlike [91] which adds in transitions based on the high-level specification. The changes made in this work are before execution unlike [61] which resynthesizes controllers during execution.

3.3 Problem Formulation

In this work, we present an system that implements a controller synthesized from a high-level task specification with ROS seamlessly. We consider manual mapping of correct-by-construction controller inputs and outputs to one or multiple ROS nodes. As described in Section 3.1, even when executing a correct-by-construction controller on a robot, the system can still create erratic execution. For instance, in Example 7, the robot can move and stop at the same time with the controller in Fig. 3.1. The resulting behavior is unknown and the robot can crash into a person. In the case of executing more than one robot under the same controller, the robots may not receive the same commands due to incorrect mapping and this may lead to unexpected execution.

We consider the following problem:

Problem 2. Given a high-level task specification and a mapping of controller inputs and

outputs to ROS nodes, provide:

(a) safety and goal-satisfaction guarantees for robot execution with ROS under user assumptions about the environment behaviors in the specification;

(b) guarantees on conflict-free message-passing to ROS topics and actions;

(c) feedback to the user about foreseeable and possible execution failure.

For our approach, we make the following assumptions about the system of interest: 1. There is only one single ROS master across multiple machines. 2. No ROS node launches another node during execution.

3.4 Preliminaries

Definition 3.4.1. Robot Operating System (ROS)

The Robot Operating System (ROS) is a robotics middleware comprising of software libraries developed and shared by researchers and hobbyists around the world. In ROS, these libraries are known as packages. To use ROS, users start by creating stand-alone programs called nodes, each denoted as n, with custom or existing ROS packages. Each node n can execute robot commands, retrieve and update sensor information or process and forward incoming data. ROS operates as a distributed system; to exchange information with other nodes, each node must connect to a ROS master. Through the master, each node can locate and communicate with another node through message-passing in three different methods:

1. **Topics:** A node n interacts with a topic T either through subscribing to information in the form of messages from the topic or publishing messages to the topic. Each topic creates a many-in-to-many-out relationship; it is a message bus that can only pass one type of message but multiple nodes can subscribe or publish messages to the topic T. We call a node that subscribes to messages from a topic the topic's subscriber while a node that publishes messages to a topic the topic's publisher.

2. **Services:** Services provide a request/reply relationship in ROS. Any node n can send service requests to a service-providing node and wait for replies as long as the two nodes are connected to the same ROS master. The duration of a request to a reply is usually relatively short. For example, in the MoveIt! [80] package of ROS, the move_group node provides an inverse kinematics service that returns joint values of a robot arm based on a given pose of the robot end effector. The time taken is usually less than a second.

3. **Actions:** Actions are treated as services that take a longer time to fulfill the request. When an action server node receives a request, the server processes the request and provides feedback to the action client node in the meantime. After the action server finishes the request, similar to a service, it returns a result of the request to the client.

   Actions function as longer-duration services but when examining the system, their connections with a node are similar to those of topics, showing as 'action_topic' in the connected system; the action client node and the action server node both subscribe and publish to the 'action_topic'.

To examine all the nodes and intertwined connections of a ROS master, a user can

examine the full system graph. The graph is known as the ROS Computation Graph, $G = \{N, E\}$, and a user can examine the Graph G with existing GUIs such as 'rqt_graph' or with APIs such as 'rosgraph'. The graph G is a directed graph; N is the set of all nodes and E is the set of all directed edges between nodes. Each edge $e \in E$ is of the form $[T, n_{start}, n_{end}]$, where T is the topic name and also the edge name, n_{start} is the starting node of the edge and n_{end} is the ending node of the edge. The graph G currently does not display service connections among nodes.

In ROS, if we have multiple homogeneous robots, we can distinguish them by defining a **namespace** in front of the topic, service or action of each robot. For example, if we have a topic, service, or action of a robot in the form '/A'; with a namespace added, it will be of the form '/namespace/A' (e.g.: '/robot1/A', '/robot2/A' etc.). With namespaces, we can distinguish among homogeneous robots easily without changing all the topics, services or actions manually. In this work, we consider robots using the same topics to be homogeneous. For example, a KUKA youBot and an Aldebaran Nao are homogeneous if they have the exact same topics.

Definition 3.4.2. High-level Task specification and Robot Controller Synthesis

Given a robot system consisting of ROS nodes, we want to provide guarantees on robot behaviors during the system execution using some existing formal methods techniques. In this work, we are interested in writing high-level task specifications and then automatically synthesizing correct-by-construction controllers from these specifications.

We give a brief overview of the process of going from a high-level task specification to synthesizing and executing a correct-by-construction controller on a robot platform. The

reader can find more details of the process in [47].

In this work, we consider a high-level task specification φ written in Structured English, such as the one shown in Spec. 1. Given a specification, we first translate it to Linear Temporal Logic formulas (LTL); using these formulas, with the controller synthesis technique in [10], we automatically synthesize a robot controller $\mathcal{A}$ if the task is feasible. Fig. 3.1 gives an example of the synthesized controller; it is a finite-state machine. Its edge labels are the input propositions (inputs) $\mathcal{X}$ of the controller and its state labels are the output propositions (outputs) $\mathcal{Y}$ of the controller.

Propositions are Boolean variables that abstract either the continuous environment behaviors for input propositions $x \in \mathcal{X}$ or the continuous robot behaviors for output propositions $y \in \mathcal{Y}$. At each state, the controller $\mathcal{A}$ first takes in a current valuation of the input propositions. Given the valuation of the input propositions and the current controller state, the controller $\mathcal{A}$ determines and moves to a next state and outputs the valuations of the output propositions at that state.

With a specification and a controller automatically synthesized, the user still cannot execute the controller on a robot platform until he or she creates a mapping from the controller inputs and outputs to some low-level programs that execute robot commands. The mapping specifies programs that provide a valuation of the inputs to the controller and programs that respond to output valuations of the controller. In this work, we map the inputs and outputs of the controller to ROS programs that either retrieve and process sensor information for inputs or command and actuate the robot for outputs.

With the mapping from the inputs and outputs of the controller to ROS programs that execute low-level commands on the actual robot, we are ready to execute the task. In the following section, we describe our framework for connecting a synthesized correct-by-construction robot controller with ROS nodes (Section 3.5.1) and methods to provide feedback to the user regarding possible problems with the mapping (Section 3.5.2).

3.5 Approach

Before we can execute the robot controller on a physical system, we need to map each input and output in the controller to one or multiple ROS programs (nodes) that retrieve sensor information or execute robot commands. In the following subsection, we elaborate on the framework of controller-to-ROS integration.

3.5.1 Mapping from Propositions to ROS Nodes

Since ROS follows a distinct communication paradigm among nodes, instead of asking existing ROS users to learn about correct-by-construction controller execution, we adapt the execution to the ROS structure. Fig. 3.2 gives an overview of the connection model between a correct-by-construction controller and ROS nodes and Fig. 3.3 shows the integration of the correct-by-construction robot controller in Fig. 3.1 with ROS.

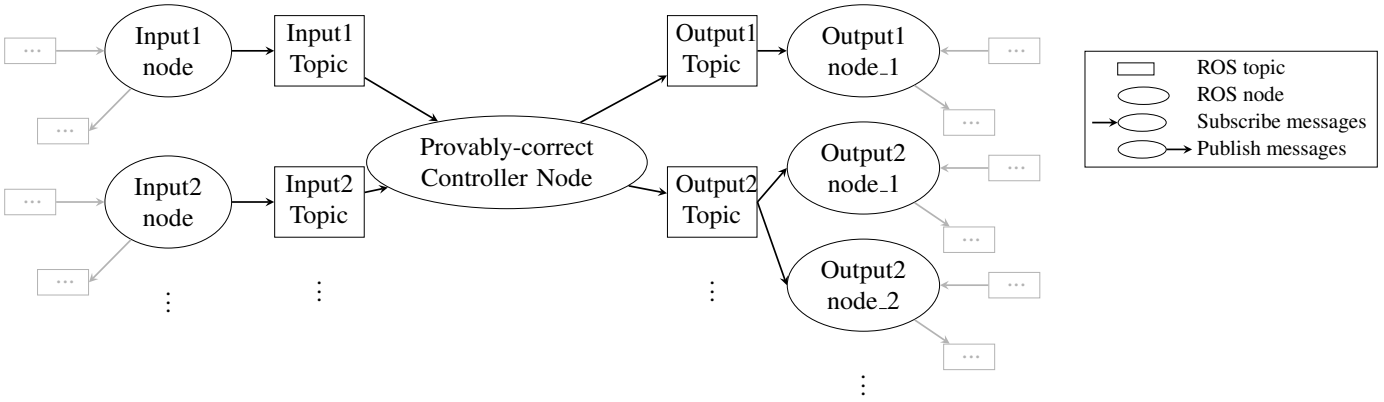


Figure 3.2: Overview of Connections to ROS Controller Node

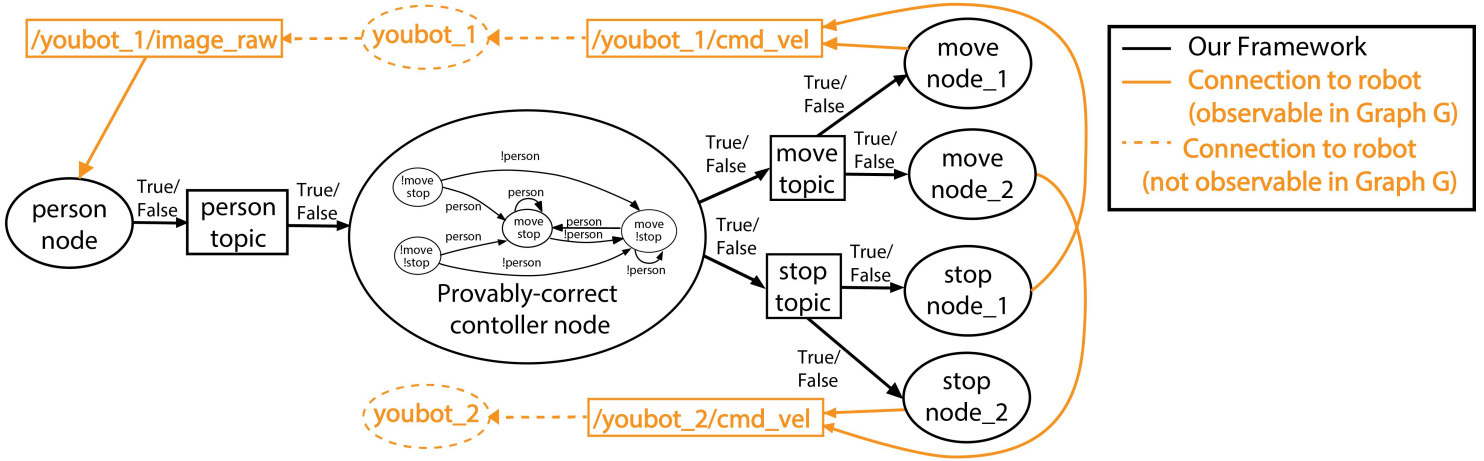


Figure 3.3: Integration of the finite state machine in Example 7 with ROS

In the structure shown in Fig. 3.2, the correct-by-construction controller forms a standalone ROS node. Each proposition in the controller corresponds to an input topic or an output topic. We refer to them as 'proposition topics'. In Example 7, the input **person** corresponds to the 'person topic', as shown in Fig. 3.3. To create a connection between the controller and ROS programs, we link each proposition topic to at least one ROS node. A user can create and modify such a mapping with a provided GUI.

A user first connects each input topic to a node; we refer to this node as an 'input proposition node'. Each input node first takes in sensor information; the node then processes and interprets the information into Boolean messages; finally the input node publishes these messages through the input topic to the controller node (See Input1 Topic in Fig. 3.2). For the input topics, each topic only receives messages from one node, i.e.: there should be only one publisher to the input topic. In Example 7, the input **person** converts sensor information from the '/image_raw' topic to decide whether the input **person** is true. The 'person node' then publishes this status to the 'person topic' and this topic is subscribed by the controller node, as shown in Fig. 3.3.

Similarly, the user also connects each output topic to one node (See Output1 Topic in Fig. 3.2) or more (See Output2 Topic in Fig. 3.2), we refer to these nodes as 'output proposition nodes'. Each output node takes in a Boolean valuation from the output topic. If the valuation is true, the output node would decide on the action commands to execute and send the commands to the robot(s); if the valuation is false, the output node is idle. We define M_{prop} to be the number of nodes connected to a topic for a proposition $prop$. For example, in Fig. 3.2, the number of nodes connected to Output1 Topic is 1 ($M_{Output1} = 1$)

and the number of nodes connected to Output2 Topic is 2 ($M_{Output2} = 2$).

For each output topic, since it can be subscribed to by one or more nodes, these nodes together can send multiple commands to multiple robots; this allows for centralized control of multiple robots under one proposition. For input topics, we refrain from allowing multiple nodes publish to the same input topic, but rather we ask the user to create multiple input propositions and decide on the instructions with those propositions at the specification level. This allows the technique in [10] to validate the instructions during controller synthesis.

In addition to communicating with the controller node, the input and output nodes can subscribe or publish to any other nodes, together with sending or receiving action and service requests and responses. In Example 7, the 'output proposition nodes' of **move** receive status messages from the 'move topic' that is published by the controller node. For the single robot case, if the output **move** is true, the 'move node_1' publishes velocity commands to the '/youbot_1/cmd_vel' topic. For the case of controlling two KUKA youbots under the same controller, if the output **move** is true, commands are sent to both robots. The 'move node_1' publishes velocity commands to youbot_1 through the 'youbot_1/cmd_vel' topic while the 'move node_2' publishes velocity commands to youbot_2 through the '/youbot_2/cmd_vel' topic. The single robot case is as shown by the orange connection on the top of Fig. 3.3 while the two-robot case is as shown by both orange connections on the top and bottom of Fig. 3.3.

The controller node executes the finite-state machine $\mathcal{A}$ synthesized from a specification φ. At runtime, it subscribes to Boolean valuation of the input propositions from the input topics and publishes the most recent valuation of the output propositions through the output

topics. These output proposition valuations are subscribed by the output nodes through their output topics.

Currently, when using ROS, a user controls a robot through one or more nodes. Each node usually contains both logical reasoning of multiple sensors and intertwined communications to different nodes, action servers and robots. Each node also subscribes to multiple topics and publishes to multiple topics. A node can easily send conflicting commands to robots and errors are often hard to debug in these scenarios. With this work, we reduce the logical reasoning inside each node and handle logical conditions in a correct-by-construction manner. We can also analyze the system and check for possible failure with these intertwined connections before execution, as described in the following section. With our approach, we trade off the number of nodes with the number of connections each node has to other nodes; increasing the number of nodes exposes connections that we then explicitly reason over.

In the following section, we focus on two of the three message-passing methods: topics and actions. For the other message-passing method, services, since the time span between a service request and response is relatively short and services are not available in the current ROS Computation Graph API, the current work does not provide feedback. However, the user can still send and receive service requests and responses in this execution model.

3.5.2 Detecting Possible Failure

With this connection model of a correct-by-construction controller with ROS nodes, we can now examine the connection among the nodes. First, we propose three ways to define possible undesirable behaviors when executing ROS nodes with a correct-by-construction controller on a single robot:

E1. Input propositions subscribing to topics published by output propositions (Section 3.5.2.2)

E2. More than one output proposition publishing to the same topic (Section 3.5.2.3)

E3. An output proposition mapping to multiple nodes and some of these nodes are sending commands to the same topic (Section 3.5.2.4)

In addition to the feedback above, we also propose two ways to detect undesirable behaviors in the case of centralized control of homogeneous robots:

C1. A topic of one robot is not connected to any proposition, while such a connection exists for the other robots (Section 3.5.2.5);

C2. A robot in the group of homogeneous robots is not controlled by any output proposition (Section 3.5.2.6);

We have shown in previous sections how E2. can lead to unexpected executions: when both output propositions **move** and **stop** are true (the yellow state in the middle of Fig. 3.1),

the output nodes of the two propositions can both publish velocity commands to the '/youbot_1/cmd_vel' topic. The mapped output node of **move** can send non-zero velocity commands while the mapped output node of **stop** can send zero velocity commands to '/youbot_1/cmd_vel'; this leads to unexpected behaviors during execution.

For E1., consider an additional line in Spec. 1 in Example 7: 'If you are sensing **privacyZone** then do **disableCamera**.' Here, the input **privacyZone** is mapped to a location-based sensor while the output **disableCamera** is mapped to the shutdown of the camera on the robot if it is true. In this case, we can observe that if the robot is in a privacy zone, the robot turns off the camera. The robot does not update the camera topic '/youbot_1/image_raw' and the robot may not notice there is a person standing in front later on. As a result, the robot can run into the person. The output **disableCamera** influences the input **person** here and this can lead to undesirable behaviors during execution.

For E3., consider only an output proposition. Here a user can map one output proposition to multiple ROS nodes (or in other words, connect an 'output proposition topic' to multiple ROS nodes), and a user can accidentally map the output proposition **move** to both the output node 'move node_1' and the node 'stop node_1', sending conflicting velocity commands to the robot. We want to avoid this scenario of sending conflicting commands within an output proposition.

In the case of two KUKA youbots, the user could have also left one youbot, youbot_2, uncontrolled without noticing it, turning a centralized robot control to a single robot control (C1. and/or C2.).

In this section, we show how we can automatically detect these problems before execution and suggest edits to the specification. Before examining any possible failure and after the user has mapped all the propositions to a corresponding ROS node, we launch all the 'proposition nodes', i.e., we start running all the ROS 'proposition nodes'. These nodes can retrieve information from other nodes but they would not execute commands on the robot at this point, as all the output propositions are false currently. With all the 'proposition nodes' started, we obtain a ROS Computation Graph G using either 'rqt_graph' or 'rosgraph'. Using the graph G, we can examine the topics or actions that each node n is subscribing or publishing to.

3.5.2.1 Retrieve the propositions-to-nodes connections

Publishing or Sending Action Requests: In the proposed framework, we represent each proposition $prop$ in the controller as a proposition topic t_{prop} connected to the controller node. Before analyzing any undesirable behaviors in the system, first we consider all 'proposition nodes' connected to the topics t_{prop} and find out the topics and actions that each 'proposition node' is reaching through publishing or sending action requests.

We denote each 'proposition node' as $n_{prop,i}$, where i in the notation $n_{prop,i}$ stands for the ith node in a total of M_{prop} nodes subscribing/publishing to topic t_{prop}. For inputs, there is only one 'proposition node' publishing to topic t_{prop} as defined by the framework. Thus, i always equals to 1 for input nodes: $M_{prop} = 1$ and $i = 1$. For outputs, we allow for multiple 'proposition nodes' subscribing to t_{prop}. Thus, i can be equal to or greater than 1, but it is always smaller than or equal to M_{prop}, the total number of output nodes connected to the

output topic of *prop*: $M_{prop} \geqslant 1$ and $i \leqslant M_{prop}$. We provide an example below to explain the algorithm to retrieve all the topics and nodes connecting to a proposition.

Consider we start with a 'proposition node' $n_{prop,i}$ connected to a proposition topic t_{prop}. This 'proposition node' $n_{prop,i}$ publishes messages to a topic '/A'; there is another node $n_{another}$ that could subscribe to the same topic '/A' and forward the messages through publishing them to another topic '/B'. To fully capture all the potential destinations of each message sent by the 'proposition node' $n_{prop,i}$, we automatically continue to traverse all the connected nodes in the Graph G until the publishing topics are not subscribed by any other nodes, or the iteration is stopped because we reach generic topics such as '/clock' or '/rosout', which every node publishes to. The algorithm also ignores nodes that are revisited.

When iterating through all the edges, the algorithm saves a dictionary of the paths for the proposition node $n_{prop,i}$ to reach a topic or node $C^{pub}_{n_{prop,i}} = \{n_1 : [n_{prop,i}, t_1, n_1], t_1 : [n_{prop,i}, t_1], \dots\}$ For example, consider the first pair in the dictionary $C^{pub}_{n_{prop,i}}$. For a message to reach the node n_1 from the 'proposition node' $n_{prop,i}$, the 'proposition node' $n_{prop,i}$ can first publishes this message to the topic t_1; this topic t_1 is then subscribed by the node n_1 and n_1 can obtain this message through t_1. We can find out this information from Graph G. From the dictionary $C^{pub}_{n_{prop,i}}$, we can retrieve a set of topics that the 'proposition node' $n_{prop,i}$ can potentially reach through publishing or sending action requests, $T^{pub}_{n_{prop,i}} = \{t_1, t_2, \dots\}$, and a set of nodes reached by a message from the node $n_{prop,i}$ through publishing, $N^{pub}_{n_{prop,i}} = \{n_1, n_2, \dots\}$. There are no duplicates in $T^{pub}_{n_{prop,i}}$ or $N^{pub}_{n_{prop,i}}$. We use these topics and paths to provide feedback in the later subsections.

Subscribing or Receiving Action Requests: Similarly, we can retrieve information about the topics and actions that each 'proposition node' $n_{prop,i}$ is subscribing to or receiving requests from. We traverse the Graph G in the reverse direction of the edges. At the end, we obtain a list of topics/actions $T^{sub}_{n_{prop,i}}$ that the 'proposition node' $n_{prop,i}$ is directly or indirectly subscribing to, a dictionary $C^{sub}_{n_{prop,i}}$ that contains the paths to different subscribe-reachable topics and nodes, and a set of nodes visited by the proposition $prop$ through subscribing, $N^{sub}_{n_{prop,i}}$.

If there are multiple proposition nodes $n_{prop,i}$ connected to a proposition topic t_{prop}, we can retrieve all the topics and nodes that a proposition is connected to through a union of the sets relating to the proposition $prop$. For example, $T^{sub}_{prop} = \bigcup_{i=1}^{M_{prop}} T^{sub}_{n_{prop,i}}$. Note that we do not remove any duplicated elements with this union operation.

With the lists of topics $T^{a}_{n_{prop,i}}$, nodes $N^{a}_{n_{prop,i}}$, and the dictionaries $C^{a}_{n_{prop,i}}$ where $a \in \{sub, pub\}$, we can now analyze the inter-connections of the nodes.

3.5.2.2 Output proposition nodes publishing to input proposition nodes

In this integration of ROS with correct-by-construction controllers, we can have output proposition nodes publish messages to topics that are subscribed by inputs proposition nodes. As described above, this can be problematic during execution.

To detect this potential failure before execution, for all output propositions, we check if the set of nodes visited by each output proposition y through publishing, $N^{pub}_{y} = \bigcup_{i=1}^{M_y} N^{pub}_{n_{y,i}}$, contains one of the input proposition nodes n_x. If the set N^{pub}_{y} contains the input proposition

node, i.e.: $n_x \in N_y^{pub}$, then the algorithm automatically saves the pair of input-output propositions $[y, x]$ and return the result that it detects a potential issue at the end.

3.5.2.3 Output propositions publishing to the same topic

In ROS, we can control a robot through publishing velocity commands to a topic, but we do not want two propositions sending velocity commands to the same topic at the same time, as the resulting behavior of the robot is unclear.

With the ROS Computation Graph G and the sets of publish-reachable topics of each proposition $T_{prop}^{pub} = \bigcup_{i=1}^{M_{prop}} T_{n_{prop,i}}^{pub}$ from Section 3.5.2.1, we can automatically detect commands sent to the same topic/action by different output propositions. To do so, we compare the set of publishing topics $T_{y_p}^{pub} = \bigcup_{i=1}^{M_{yp}} T_{n_{yp,i}}^{pub}$ of one output proposition y_p with the set of publishing topics $T_{y_q}^{pub} = \bigcup_{i=1}^{M_{yq}} T_{n_{yq,i}}^{pub}$ of another output proposition y_q for $q \neq p$. If the intersection of the sets $T_{y_p}^{pub}$ and $T_{y_q}^{pub}$ is not empty (e.g.: $T_{y_p}^{pub} \cap T_{y_q}^{pub} = \{t, \dots\}$), then that means both propositions y_p and y_q can potentially publish messages to the same topic simultaneously during execution. This can create erratic and undesirable robot behaviors; we notify the user and automatically generate mutual exclusion specifications for these propositions that the user can add into the specification. Note that as each 'output proposition node' only sends commands to the robot when that output proposition is true, mutual exclusion makes sense here; the new specification can prevent conflicting commands from being sent to the robot during execution.

First, we automatically save all the pairs of concurrent-topic-access propositions with

the corresponding topic, $\{\{t, p, q\}, \dots\}$. Once we have found all the possible concurrent topic accesses, we can automatically suggest modification to the high-level task specification in the form of Structured English sentences.

For instance, in Example 7 for the single robot case, when we detect that the output propositions **move** and **stop** both publish to the same topic '/youbot_1/cmd_vel', we can suggest the user to add in a sentence saying that '**move** and **stop** are never true together'. In the form of Structured English, it is 'always (not **move** and **stop**) or (**move** and not **stop**) or (not **move** and not **stop**)'.

3.5.2.4 An output proposition sending conflicting commands to the same topic through multiple nodes

For each output proposition $prop$, since a user can connect multiple 'proposition nodes' $n_{prop,i}$ to an output topic t_{prop}, these nodes $n_{prop,i}$ may publish different commands to the same topic t when the output proposition $prop$ is true. This can send conflicting information to a robot within the same output proposition $prop$ and lead to unexpected behaviors. With the mapping created by the user, we can check for such an incident.

The set T^{pub}_{prop} is the union of all the sets of topics $T^{pub}_{n_{prop,i}}$ of each node $n_{prop,i}$ mapped to the output proposition $prop$. For each set $T^{pub}_{n_{prop,i}}$, there are no duplicates of any topic in the set. To analyze the possible error described here, we can check if there is a duplicate in the set of publishing topics T^{pub}_{prop} for each output proposition $prop$. If there exists more than one copy of a topic t in the set T^{pub}_{prop}, then this must be from two different sets of $T^{pub}_{n_{prop,i}}$.

This indicates that more than one node can talk to the same topic for the proposition *prop* at the same time.

For example, if the 'move topic' is accidentally mapped to both the 'move node_1' and the 'stop node_1', then both nodes can publish commands to the topic '/youbot_1/cmd_vel' at the same time. There would be one copy of '/youbot_1/cmd_vel' in the set $T^{pub}_{n_{move,1}}$ and another copy of '/youbot_1/cmd_vel' in the set $T^{pub}_{n_{stop,1}}$. In the set $T^{pub}_{move} = T^{pub}_{n_{move,1}} \cup T^{pub}_{n_{stop,1}}$, there would be two copies of the topic '/youbot_1/cmd_vel': {'/youbot_1/cmd_vel','/youbot_1/cmd_vel',...}. We can notify the user of this potential undesirable behavior before execution.

3.5.2.5 Multiple Homogeneous Robots: check if a topic of a robot is left behind

In the case where we control a group of homogeneous robots with one correct-by-construction controller under our paradigm, given a list of robots R by a user, we can check if a robot $r \in R$ or a topic of the robot r is not connected to the controller.

Following the namespace convention in ROS, we assume that if we are controlling a group of homogeneous robots, then for the same ROS topic on each robot, it will be of the form '/robot1/topic1', '/robot2/topic1' etc. We denote them as 'robot topics'.

For each output proposition *prop*, we start by finding all the 'robot topics' in the set T^{pub}_{prop}, i.e., the topics that the output proposition nodes are sending commands to the robots. For all the robot topics '$/r/t$' found, we first extract all the distinct suffixes '$/t$'. For each distinct suffix '$/t$' found, we check if we can find all robot topics '$/r/t$' of this suffix '$/t$'

for all the robots $r \in R$ in the set T^{pub}_{prop}. If we can find $|R|$ robot topics for this suffix '$/t$', that means this output proposition $prop$ commands all the robots through the same topic and type of message. Otherwise, for this proposition $prop$, we are not commanding all the robots R through the same topic and we may not be able to control all of them at the same time. We provide this feedback to the user and the user can verify these mappings are as intended before execution.

For instance, in Example 7, if the user wants to control two KUKA youBots at the same time when the proposition **move** is true, but in the set T^{pub}_{move}, we can only find '/youbot_1/cmd_vel', the velocity topic of youbot_1, but not '/youbot_2/cmd_vel', the velocity topic of youbot_2. With our analysis, we can point out this case and verify with the user.

3.5.2.6 Multiple Homogeneous Robots: check if none of a robot's topics is connected to a proposition

With this, we can also check if a robot $r \in R$ is completely left out in the motion or action execution, i.e., if none of the topics of this robot r is connected to an output proposition of this controller.

To do so, we find all the robot topics in all the sets of T^{pub}_{prop} for all output propositions $\mathcal{Y}$. For the set of all robot topics found, we check if there exists at least one robot topic of the robot r, i.e., a robot topic with a prefix '$/r$'. If there does not exist a robot topic with a prefix '$/r$', then we notify the user that a robot may be excluded in this execution.

For example, if youbot_3 is in the robot list R provided by the user but we cannot find any robot topics of youbot_3 in T^{pub}_{prop}, we notify the user.

The reader can find the implementation of our approaches in our ROS package online[1]. A user can create a mapping and execute a controller on a robot platform using our plugin in this package. This is shown in the following section as we walk through two examples to demonstrate our approach.

3.6 Example

In this section, we demonstrate our framework and how we can detect undesirable behaviors before execution with two different examples: in the first example, a KUKA youbot is conducting a clean and patrol task; in the second example, four Sphero SPRK robots are responding to different sensor inputs. The reader can find a video of an analysis together with an execution of the two examples online [2].

3.6.1 Clean and Patrol Example

In this example, a KUKA youbot with an arm conducts a clean and patrol task in the workspace shown in Fig. 3.4. The specification is as written in Spec. 2.

Example 8. *The robot patrols all the outer regions,* ***topLane****,* ***rightLane****,* ***bottomLane*** *and* ***leftLane****, if it is not holding an object (line 4 in Spec. 2). If the robot sees an object, it will stop and pick up the object (line 1-3). Then it will head to* ***rightGround*** *and drop off the*

[1] *https://github.com/VerifiableRobotics/LTL_stack*
[2] https://youtu.be/E_GpZTdlx_s

object (line 5- 8).

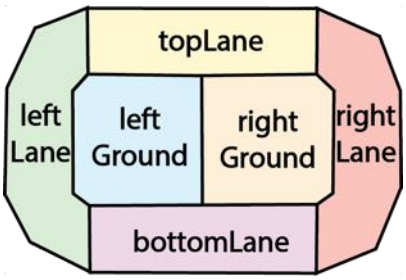


Figure 3.4: Map for Example 8

Specification 2 Clean and Patrol Specification

1 **holdingObject** is set on **finished pickup** and reset on **finished drop**.
2 do **pickup** if and only if you are sensing **object** or **sawObject** and you are not activating **holdingObject**.
3 **sawObject** is set on **object** and reset on **finished pickup**.
4 If you are not activating **holdingObject** then visit **topLane**, **rightLane**, **bottomLane** and **leftLane**.
5 If you are activating **holdingObject** then visit **rightGround**.
6 do **drop** if and only if you are activating **holdingObject** and you have **finished rightGround**.
7 do **stop** if and only if you are activating **holdingObject** and you have **finished rightGround**.
8 infinitely often **drop** and **finished drop**.

The propositions **finished pickup** and **finished drop** in Spec. 2 are known as the completion propositions [75]. These propositions keep track of the status of actions – in this case the output propositions **pickup** and **drop** respectively; the completion proposition turns true when the corresponding action is completed. For example, **finished pickup** should turn true after the robot completes its **pick** action.

We leverage the synthesis technique in [10] to check if the task is feasible that we automatically synthesize a correct-by-construction controller with the specification in

Spec. 2. In this case, a correct-by-construction controller is successfully synthesized.

3.6.1.1 Proposition mapping

Before executing the controller on a KUKA youbot, we connect the controller with ROS programs that retrieve sensor information or send commands to the robot. In our online repository, we include a Propositions Mapping and Analysis Plugin that allows the user to create mapping from propositions to ROS nodes that retrieve sensor information or execute robot commands. For each 'proposition node', the user specifies a topic in the node that corresponds to the 'proposition topic' t_{prop}.

Before mapping propositions to ROS nodes, first the user launches all the nodes to interface with the propositions; the user also provides a list of propositions for mapping by loading a specification file in the format of .slugsin [24] (A in Fig. 3.5) into the Plugin. With the list of propositions, the user can either supply an existing mapping file (B in Fig. 3.5) or create a new mapping from scratch.

Based on the nodes connected to the current ROS Master, for inputs, the user uses a drop-down box in the middle (C in Fig. 3.5) to assign a 'input proposition node' to the input. Once the user selects a 'proposition node', the user also specifies a topic in the node that serves as an input topic to communicate with the controller node with the drop-down box on the right (D in Fig. 3.5). For outputs, the user can click on the '+' button to assign new 'output proposition node' with a drop-down box similar to the case of inputs. The user can assign multiple 'output propositions nodes' and remove each of them with the

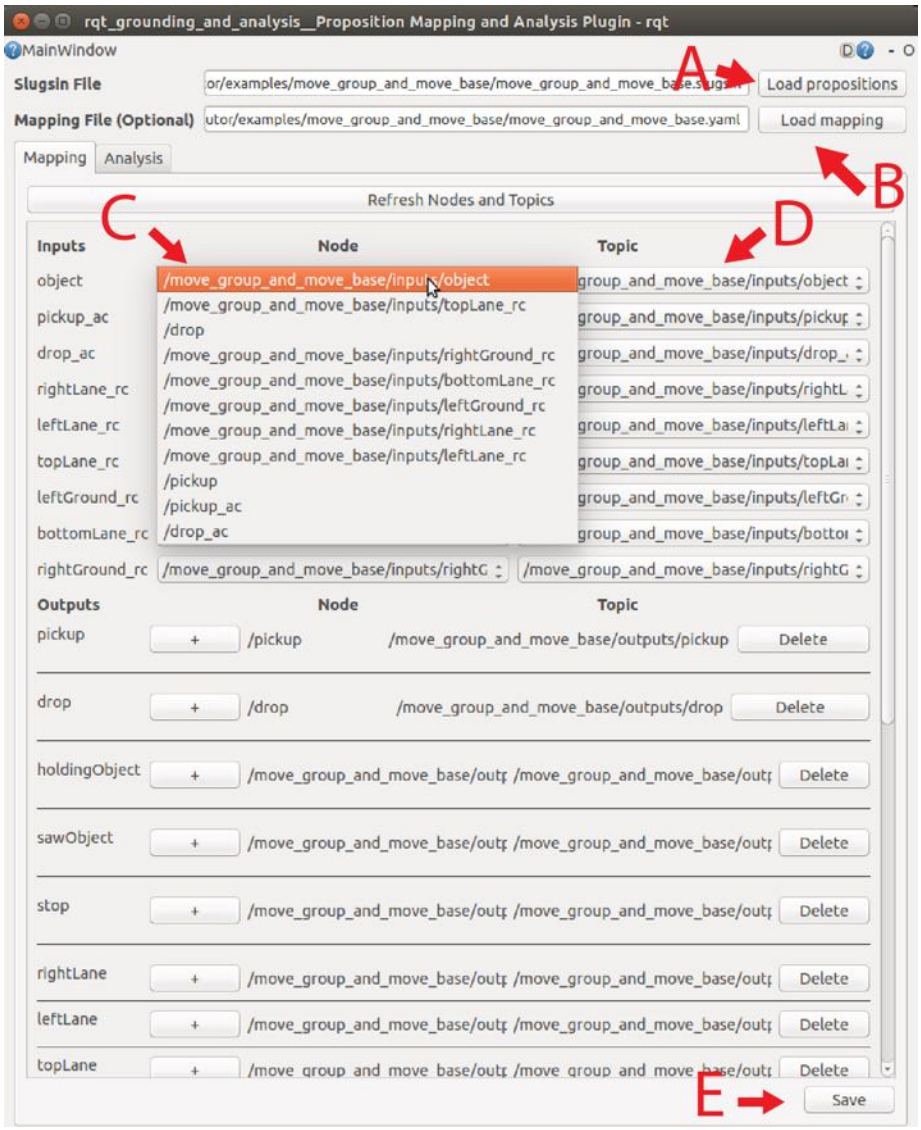


Figure 3.5: The Proposition Mapping GUI included in our package. We are mapping propositions to ROS nodes and topics for Example 8 here.

'delete' button right next to each assignment. Once the mapping is done, the user can save the mapping at the end (E in Fig. 3.5).

For this task, we localize the KUKA youBot using a Vicon Motion Capture System. All the nodes in the ROS structure can subscribe to the robot location leveraging the package vicon_bridge [13]. We have the output region propositions, e.g., **leftGround**, **rightGround** etc., mapped to nodes that drive the robot to different regions using the navigation stack [79]. We keep each node simple and separate the motion planning to different nodes here, as we want to avoid having a node that drives a robot to different waypoints; in that case, we cannot analyze the logic within a node; the intertwined connections and complex reasoning in a node can also lead to erratic execution. We map the **pickup** and **drop** propositions to nodes that plan arm trajectories using MoveIt! [80]. The proposition **pickup** also sends velocity commands to move closer to the object before picking up the object. For object detection, we use an RGBD camera mounted in front of the youBot together with the AprilTag library [68] available in ROS to detect objects. The proposition **stop** sends velocity commands of zero when it is true. With all the propositions mapped to ROS programs, we can now analyze the ROS connections.

3.6.1.2 ROS structure analysis

Once we obtain a mapping from the user, we take a snapshot of the current ROS Computation Graph G. With the graph G, we use the algorithms described in Section 3.5.2.2 to 3.5.2.4 to analyze the node connections. The results are given as follow:

Output proposition nodes publishing to input proposition nodes (Section 3.5.2.2):

Sensor Retrieving Info from Publishers

Output	Input	Output-to-input Chain
pickup	pickup_ac	(/pickup) -> [/move_group_and_move_base/outputs/pickup_status] -> (/pickup_ac)
drop	drop_ac	(/drop) -> [/move_group_and_move_base/outputs/drop_status] -> (/drop_ac)

Figure 3.6: Analysis result of output propositions publishing to input propositions for Example 8

Fig. 3.6 gives the result of using the Section 3.5.2.2 approach with the current graph G. For each row in the table, the leftmost column displays the output proposition that publishes to the input proposition; and the input proposition is shown in the middle column. The rightmost column displays one possible path from the output node to the input node in the ROS computation graph G. For the entries in the Output-to-input Chain column, the name with parentheses '()' stands for a node, (node_name), while the name with box brackets '[]' stands for a topic, [topic_name].

As shown in Fig. 3.6, with the current mapping, we can see that the output proposition **pickup** publishes its most-up-to-date status to the input proposition **finished pickup** during execution, and similarly for **drop** and **finished drop**. In this case, this connection is desirable as we want to know the arm actuation status before continuing the robot movement.

Output propositions publishing to the same topic (Section 3.5.2.3):

Both Fig. 3.7 and Fig. 3.8 highlight some results of the algorithm described in Section 3.5.2.3. With the youBot, we can control the robot movement by publishing velocity commands to the topic '/cmd_vel'. As shown in Fig. 3.7, the corresponding nodes of the output region propositions are all publishing velocity commands to the robot through

Figure 3.7: Analysis result of output nodes publishing to the robot velocity topic for Example 8

the navigation stack. The execution is undefined if more than one of the nodes publish velocity commands at the same time. In our previous work [47], we manually add in a mutual exclusion specification of the region propositions to resolve this issue, but with the framework here, we can automatically reason about this through an analysis of the ROS Computation Graph G.

Besides the output region propositions, we also find out that the output propositions **stop** and **pickup** are both directly publishing to the velocity topic. A user could not easily find out this potential conflict before this work.

With this feedback, we suggest the addition of a mutual exclusion specification 'The region propositions, **stop** and **pickup** are always mutually exclusive' (always (**stop** and not

pickup and not **leftLane** and not **rightLane** . . .) or . . .).

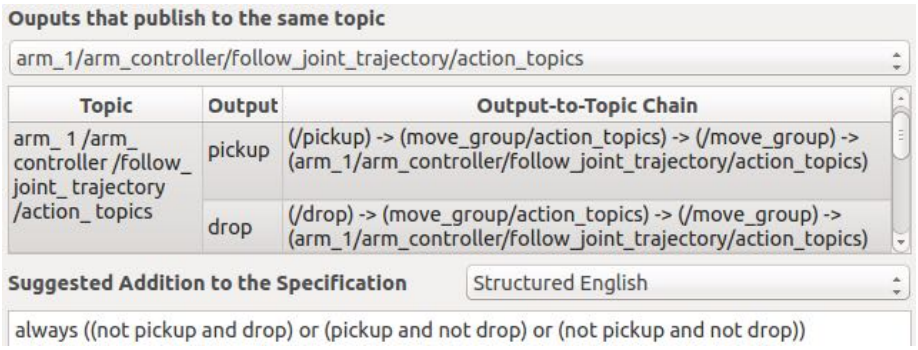


Figure 3.8: Analysis result of output nodes publishing to the youBot's arm controller for Example 8

Besides the velocity topic, the **pickup** and **drop** propositions can publish to the youBot's arm controller at the same time, as shown in Fig. 3.8. In this case, we can suggest the addition of '**pickup** and **drop** are always mutually exclusive' to the specification.

An output proposition sending conflicting commands to the same topic through multiple nodes (Section 3.5.2.4)

If the user has accidentally mapped both the 'pickup node' and the 'drop node' to the output **pickup**, then the output **pickup** can send conflicting information to the robot when **pickup** is true. In Fig. 3.9, we can see that this error is detected and the user can catch this mistake before execution.

3.6.1.3 Execution

The user modifies the specification φ based on the analysis and suggestions and launches the controller node with an updated specification and controller. During execution, the

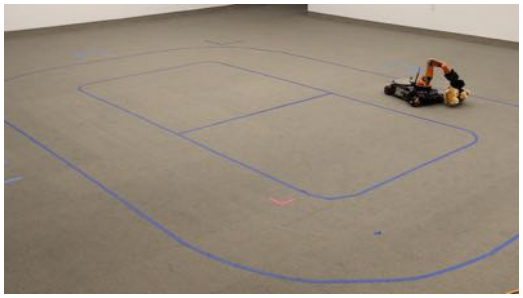
Proposition nodes of an ouput publishing to the same topic

ıut Propos	Topic	Node	One existing Chain
pickup	/arm_ 1 /gripper_ controller /position_ command	/pickup	(/pickup) -> [/arm_1/gripper_controller/position_command]
		/drop	(/drop) -> [/arm_1/gripper_controller/position_command]

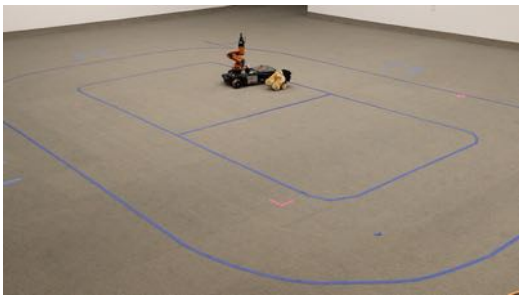
Figure 3.9: Both the 'pickup node' and the 'drop node' are mapped to the output **pickup**

robot patrols the outer regions (Fig. 3.10a). When the robot senses an object and picks up the object (Fig. 3.10b), it heads to **rightGround** and drops off the object (Fig. 3.10c). The task continues (Fig. 3.10d). None of the mutually excluded propositions publishes to the same topic at any time.

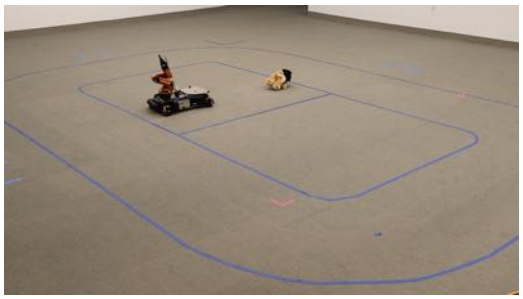
(a) The youBot starts in **topLane** and patrols the outer regions.



(b) The youBot sees an object in **bottomLane** and it pickups up the object.



(c) The youBot drops the object at **rightGround**.



(d) The youBot continues its execution.

Figure 3.10: A youBot performing its task as described in Example 8.

Figure 3.11: Four Sphero SPRKs for Example 9

Specification 3 Homogeneous Robot Control Specification

1 if you are sensing **leftSignal** then do **moveLeft** and **turnPurple**
2 if you are sensing **rightSignal** then do **moveRight** and **turnBlue**
3 if you are sensing **forwardSignal** then do **moveUp** and **turnGreen**
4 if you are sensing **backwardSignal** then do **moveDown** and **turnYellow**
5 if you are sensing **stopSignal** then do **stop** and **turnRed**
6 if you are not sensing (**leftSignal** or **rightSignal** or **forwardSignal** or **backwardSignal** or **stop**) then do **turnWhite**

3.6.2 Homogeneous Robots Example

In this example, we control a group of homogeneous robots. We are using four Sphero SPRK robots as shown in Fig. 3.11. These robots can roll in all directions and change color. Their task is as follows:

Example 9. *Control four robots as a group. The robots should respond to different commands together, including: move left, move right, move forward, move backward, stop and do nothing.*

The task specification is shown in Spec. 3.

3.6.2.1 Mapping

In this example, all the input propositions end with **Signal** and they are mapped to sensor nodes that determine if there is a signal using the AprilTag library. We use each output proposition to control all four robots. All the propositions that start with **move** or **turn** are output propositions. For each output proposition, we can map to multiple 'output proposition nodes'; an example is shown in Fig. 3.12. We use namespaces here to distinguish the homogeneous robots; the namespaces are '/sphero_wpw', '/sphero_ggw', '/sphero_rgw' and '/sphero_wpp'.

Outputs		Node	Topic
move_left	+	/spheros/outputs/sphero_ggw/move_left	/spheros/outputs/move_left
		/spheros/outputs/sphero_wpp/move_left	/spheros/outputs/move_left
		/spheros/outputs/sphero_rgw/move_left	/spheros/outputs/move_left
		/spheros/outputs/sphero_wpw/move_left	/spheros/outputs/move_left

Figure 3.12: Mapping for an output proposition **move_left**

3.6.2.2 Analysis

Since the output propositions that start with **move** (similarly for **turn**) can all send velocity commands (color commands) to the robots (See Fig. 3.13), we should modify the specification to state that the propositions starting with **move** are mutual exclusive.

With this modification, however, the specification is unrealizable. In this case, we can use to the technique in [74] to analyze and modify the specification. With [74], we find out the outputs starting with **move** are affected by the inputs ending with **Signal**. In order

to synthesize a controller, we should also assume that the inputs ending with **Signal** are mutual exclusive. With this addition, the specification is realizable. We can now move on to the analysis among the four robots.

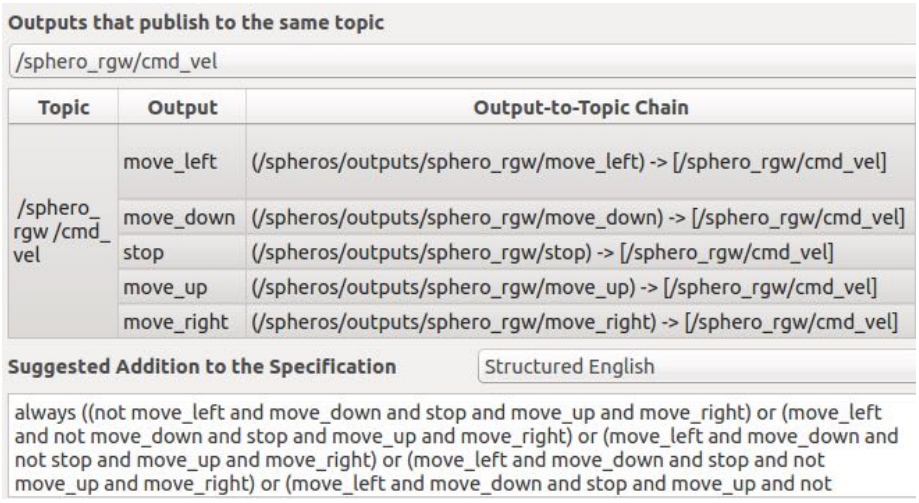


Figure 3.13: Analysis result of output nodes publishing to the robot velocity topic for Example 9

Before any analysis, first we ask the user to specify all the robots in this task (The **Robots** row in Fig. 3.14) and the robot topics that we can ignore in this analysis (The **Robots Topics to Ignore** row in Fig. 3.14). In this example, there are four robots in total. We are ignoring all the sensors on the robot, since the task does not depend on the robots' sensors.

With a defined proposition mapping, a list of robots and an optional list of topics to ignore given by the user, we analyze and highlight any robot topics or robots that are not included in the execution as defined by the current mapping.

In Fig. 3.14, we can see that one of the SPRK robots, *sphero_wpw* is not connected

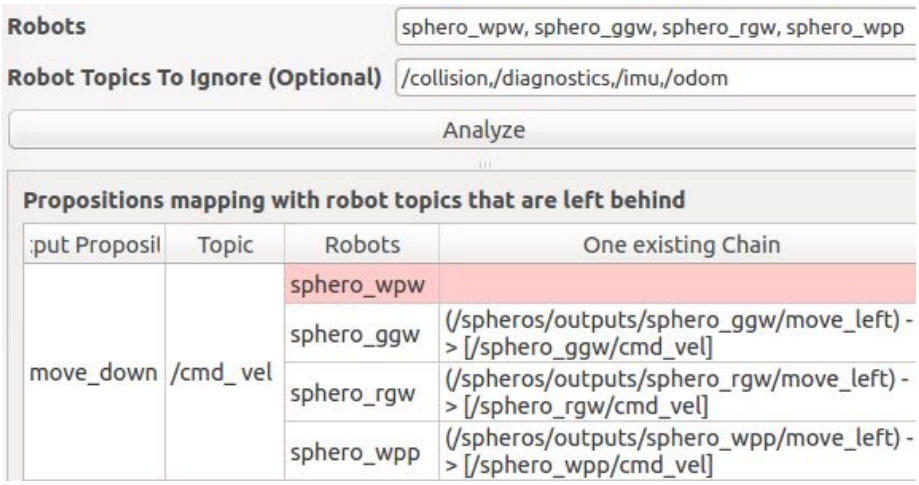


Figure 3.14: Analysis with Section 3.5.2.5

to the proposition **move_down** with the topic '/cmd_vel' , while all the other robots in the given list are. This serves as a feedback to the user and the user can then decide if he or she wants to modify the mapping.

If one robot in the robot list is completely ignored in the current mapping, we can also detect that and notify the user. As shown in Fig. 3.15, there are five robots in the list with *sphero_wrb* added, and we can check and notice that *sphero_wrb* is completely ignored in the current mapping.

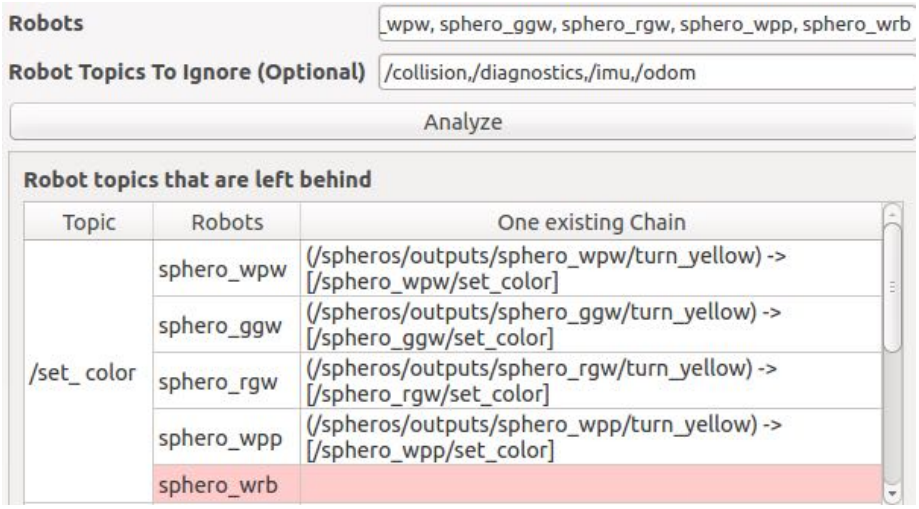


Figure 3.15: Analysis with Section 3.5.2.6

This example is shown in the video online, with four Sphero SPRKs responding to different sensor inputs.

3.6.3 Discussion

In this section, our examples demonstrate the execution of our framework with a single robot and with a group of homogeneous robots. Our approach detects potential conflicts that are not originally captured in the high-level task specification. Since the specification does not include these conflicts, the synthesis process cannot detect the conflicts and a controller is synthesized. Even though the synthesized high-level controller is provably-correct, the execution can still be different from the user expectation due to the low-level conflicts and the task outcome is unknown. With our approach, we provide warnings and suggest specification changes on these potential conflicts such that users can incorporate

the information about the low-level connections into the high-level task specification.

As shown in Example 8, our approach automatically finds out the robot can only head to one region at a time. This is previously a user insight but now we can obtain such an information automatically by inspecting the low-level connections. Our approach furthermore detects that other low-level programs can be sending commands to the robot at the same time the robot is heading to a region. Previously, this is only found out during execution; the user would stop the robot then and inspect the erratic robot behavior. For a group of robots, we can detect the robots are not in sync in Example 9 leveraging the connections of the low-level programs. With our framework, we detect problems and suggest changes before they arise during execution.

In the process of constructing and using this framework with ROS, we face some challenges. To start, the mapping from propositions to ROS nodes is still manually done by the user and this can be cumbersome if there are a lot of propositions. However, even though the mapping is manual, we have made it more streamlined and explicit with the work here that the user does not have to inspect the ROS Computation Graph G for mapping; a user can create a mapping with our provided GUI within a short period of time. A lot of ROS packages such as MoveIt! and the navigation stack are powerful, but we also spent a substantial amount of time tuning the parameters in these packages to get them working with the youBot. These packages are powerful but they must be customized for different robots. Lastly, a user still needs to know about the robot platform to ensure correct behaviors. For example, a robot without an arm would not finish the task described in Example 8 and currently we cannot detect such a failure.

3.7 Evaluation

In this section, we qualitatively evaluate our approach independent of the examples above.

In this work, we propose a framework to seamlessly execute a controller synthesized from a high-level task specification with low-level controllers in the form of ROS nodes. With this framework, we can analyze the connections among the low-level controllers for potential conflicts and propose changes to the high-level specification.

Compared to the normal approach of writing a single ROS node with many intertwined commands and communication for a robot task, our approach breaks down the node into multiple smaller nodes and allows for independent node analysis. Our approach does increase the number of low-level ROS nodes which can, in turn, increase network traffic. Although ROS can handle hundreds of nodes in normal cases, bad connectivity among nodes can hinder task progress and result in delays that are more severe than that of a single ROS node. However, by separating the logic from the commands in the single ROS node with our framework, we can analyze the connections among the nodes and make modifications to the specification. An analysis of the connections inside a single ROS node is close to impossible before. In the single node programmed by the user, the logical reasoning and the commands to the robot are usually inseparable, making it hard to analyze and extract potential command conflicts.

As for the analysis, even though we are suggesting changes based on the interactions among the output nodes, an analysis of the interactions among the input nodes is still

missing in this work. This analysis would be good for a better representation of the connections among the low-level controllers. Creating such an analysis is challenging as we cannot determine the internal data processing of each input node solely based on the ROS Connection Graph. The Graph only indicates node connections, but it does not provide the type and values of the data in and out of a node. The Graph is sufficient for an output node analysis, for which the actual data is not necessary, but it is missing the data that is critical for an input node analysis. For the input analysis, we need the data to determine the relationship between sensor data and input valuations. This information is then used to detect conflicts and suggest changes. Other structures that describe the input nodes are needed to extract such information for analysis.

3.8 Conclusion

In this work, we propose a framework for seamless integration of correct-by-construction controllers with ROS. The subscribe-publish message-passing method of ROS matches with the input-output paradigm of correct-by-construction controllers. Yet, failure can arise in the low-level execution with ROS when using these correct-by-construction controllers: the connection of a high-level controller to low-level programs is not inspected and even though the high-level controller is correct-by-construction, its interaction with the low-level programs may not be; the programs can send conflicting commands to a robot and this can lead to unexpected robot behaviors. In this work, we describe approaches to detect possible failure and provide feedback to the user using such a system.

ROS has enabled and sped up both robot software and hardware development, and it

will continue to increase its exposure to the public in the future. We lay out the starting point of providing guarantees and feedback in robot execution with ROS. Challenges and future development include using our framework with multiple ROS masters, integrating our framework with SMACH and providing analysis with the robot physical constraints and its sensors considered. For task execution with multiple robots, an analysis on task execution with a group of heterogeneous robots would be useful as well.

CHAPTER 4

RESILIENT, PROVABLY-CORRECT, HIGH-LEVEL ROBOT BEHAVIORS

4.1 Introduction

When developing robot controllers, users make assumptions about the operating environment of the robot. Such assumptions are typically kept implicit when manually coding a controller, while they are made explicit in the automatic synthesis of controllers. For example, in a pick-up task, one might assume that an object is always available for pickup. If such assumptions are violated at runtime, two problems usually arise: (i) the controllers do not accommodate unexpected environment behaviors and hence, the execution outcome is unknown, and (ii) the controllers do not provide feedback to the user. In this work, we consider controllers synthesized from high-level specifications, with explicit assumptions about the environment. We increase the robustness of these controllers with respect to unexpected events so that the robot maintains its behavioral guarantees, whenever possible. When unrecoverable unexpected situations arise, we provide feedback to the user by leveraging the work in [74].

Automatic controller synthesis from high-level task specifications has gained popularity in the robotics community in recent years (e.g. [6,7,34,39,41,44,45,73,85,93]). Compared to the tedious and error-prone nature of manually writing robot controllers, these synthesized controllers provide correctness guarantees, i. e., the generated controller satisfies all requirements in the specification given by the user. We call this property provably-correct and these controllers provably-correct controllers.

To synthesize provably-correct controllers, some formulations consider specifications that capture both the robot and its environment behaviors (e.g. [44, 45, 93]). Similar to those works, each specification considered here is separated into two parts: environment assumptions and robot specification [10]. To guarantee the desired behavior of the robot, the environment assumptions must be satisfied when executing the synthesized controller. The controller synthesis process creates robot behaviors for any "allowable" environment behaviors, based on the environment assumptions. If the environment violates its assumptions at runtime, then the controller may violate the robot specification.

Example 10. *Consider a task where a robot is required to grasp a block and cut it in half by pushing it towards a table saw. In a manually-coded controller, an engineer may assume that a block is always at the pickup location when the robot actuates its pickup action. However, if there are no objects at the pickup location and the engineer does not check the assumption during execution, the robot will send its "block", or its hand, into the saw and destroy it. In an automatically synthesized controller, the user may explicitly state in the environment specification that a block is present when the robot actuates its hand; when executing, if the robot is asked to pick up a block but there isn't one, then the assumption is violated and this results in unknown robot action.*

In this work, we present a framework to automatically detect and recover from environment behaviors that violate the environment assumptions, at runtime. This framework consists of three complementary methods that restore system correctness guarantees, such that the robot can continue its task as defined in the robot specification after the violation. We focus on how a robot resolves conflicts with its environment or one other collaborative agent in the workspace; multi-agent conflicts are out of the scope of this work. One

component of our framework incorporates possible runtime anomalies offline through the controller synthesis process (*Recovery* - Section 4.6), while the other two resolve any issues online during the controller execution (*Environment Characterization* - Section 4.7 and *Integrative Negotiation* - Section 4.8).

The *Recovery* approach in Section 4.6 synthesizes a "forgiving" controller that tolerates temporary anomalies, i.e., environment behaviors inconsistent with the environment assumptions, when executing the controller. In Example 10, with the Recovery approach, the robot would wait until the block is available and then pick it up.

As the recovery approach does not guarantee eventual progress towards the robot goals when environment assumption violations persist, we furthermore introduce an *Environment Characterization* approach. It rewrites the environment safety assumptions by incorporating newly-observed environment behaviors. The specification now allows more environment behaviors; if the robot can still satisfy the task with the new assumptions, the robot can continue its task. In Example 10, the specification would be updated during execution and the robot would stand by until the block is present and then actuate its pickup action.

If the environment includes other collaborative robots in the same workspace, then the *Integrative Negotiation* approach in Section 4.8 attempts to modify the environment behaviors, specifically the other robot's behavior, through communication with the other robot. Consider Example 10 but now the block is placed at the pickup location by another robot. To ensure the block is ready, the robot assumes its counterpart leaves a block at the pickup area when the area is empty. However, if its counterpart is idle at runtime thus violating the assumption, the robot can then initiate this approach to recover its assumption.

This paper expands on the work outlined in [86] and [88] in several directions. First, the Recovery approach updated from [86] now accommodates more violations of environment assumptions and allows the robot to temporarily stop progressing to its goals until the violation is resolved. The approach still ensures that if the anomaly is temporary, the robot can recover and head towards its goals. In [86], the robot controller fails when the robot cannot guarantee one of its goals.

Second, this paper leverages the activation-completion paradigm described in [75]. When combined with the Environment Characterization approach in [86], we can capture relationships of environment behaviors and robot behaviors: the updated environment assumptions can now include not only environment behaviors but also robot status such as its location and its completed actions, thereby refining the assumptions; previously without the activation-completion paradigm, the approach in [86] could only capture the environment behaviors but not any robot status. For example, in [86], the approach could only capture that a light is on, but here it can capture that a light is on when the robot is sensed to be in region A.

Finally, the improved Integrative Negotiation approach based on the work in [88] now allows two robots in conflict to both accommodate the other's task and proceed to their goals, ensuring no future violations would be caused by the same conflict. Consider the example above again: robot A assumes that robot B places a block at the pickup area when it is empty, while robot B assumes that robot A never enters its region. Robot B can violate robot A's assumption and not place blocks during execution. In that case, the Integrative Negotiation is triggered; both robots incorporate and satisfy the other's assumption after

negotiation. Robot A will stay away from robot B while robot B will replace blocks at the pickup area. This violation will not happen in the future. In [88], only one robot accommodates the other robot after negotiation, and the other robot can create conflicts again.

The paper is organized as follows: In Section 4.2, we review related work. In Section 4.3, we define preliminaries and the specification for Example 10. In Section 4.4, we define the problem and give an overview of the three approaches in Section 4.5. These approaches are described in Sections 4.6 (Recovery), 4.7 (Environment Characterization) and 4.8 (Integrative Negotiation). In Section 4.9, we discuss the computational implications of the three approaches. In Section 4.10, we illustrate the approaches with examples. We provide an evaluation of our approaches afterwards in Section 4.11 and finally, we summarize our work in Section 4.12.

4.2 Related Work

4.2.1 Planning

The work on controller synthesis from high-level specifications shares some of the same objectives as work in the AI planning community. Relevant planning approaches can be separated into two types:

• Related **Offline Planning** approaches include conformant planning and contingency planning. Conformant Planning constructs plans that work in all possible scenarios of

a partially-observable or unobservable world. Related works leverage model checking techniques to validate plans against specifications in an unobservable domain [16, 17, 29]. Schoppers synthesizes a plan for a robot with one goal using the Universal Plans approach [78].

Contingency Planning "constructs plans that can be expected to succeed despite unknown initial conditions and uncertain outcomes of nondeterministic actions" [71]. The authors of [21] find contingency plans with failure risk within a known bound. Others tackled contingencies by providing a variety of online recovery approaches for different exceptions [35] or planning multiple contingency paths simultaneously to account for future uncertainties for autonomous vehicles [33].

In our work, robots operate in a fully observable world unlike conformant planning. Similar to contingency planning, we create controllers that guarantee task completion from a set of possible initial conditions and deal with nondeterministic action outcomes. The conceptual difference between the two is that in contingency planning the nondeterminism is over controlled actions, all outcomes are possible, while in our work, the nondeterminism is over uncontrolled environment events. Some events may be assumed not to happen, and the controller is synthesized based on a worst-case analysis; if a controller exists, the task is guaranteed to succeed.

• **Online Replanning** responds to unexpected events during execution. Some researchers repair the current plan online through either adding or removing actions [83]. Others use failure in the form of transition matrices to modify to the planning instance [19]. In our work, we also use failures, violations of environment safety assumptions, to modify robot controllers by updating the specifications.

Our work shares similar ideas to contingency planning and online replanning; Our approach combines those problems to create a system that is robust to unexpected events. Though both AI planning and controller synthesis focus on creating a plan/controller, our work synthesizes controllers that can provide correctness guarantees for user specifications during execution and replan for unexpected events online, while most of the works in the AI planning address the correctness problem and the robustness against anomaly problem separately. Furthermore, the specifications typically addressed in contingency planning and online replanning are of the form "reach a goal state" while this work considers specifications with multiple goals and safety constraints.

4.2.2 Negotiation and Collaboration

Researchers define negotiation as "a means for agents to communicate and compromise to reach mutually beneficial agreements" [95]. Some researchers model multi-issue negotiation under time constraints and incomplete information [27]; others optimize the acceptance policies in a negotiation [5]. Researchers also use game theory to analyze automated negotiation among agents with partial information and shared tasks [97]. In collaboration, researchers reduce the plan repair problem for coordination of decentralized agents to a multi-agent planning problem that minimizes communication overhead [42]. In this work, our Integrative Negotiation approach is inspired by existing negotiation and collaboration approaches. Our approach focuses on the exchange of information so that each robot can resolve conflicts with its environment and the robots in our work are not sharing tasks.

4.2.3 Controller Synthesis

In controller synthesis from temporal logic specifications, researchers have considered different types of failures and improvements on controller execution:

• **Synthesis Failure:** Some authors provide explanations on why controller synthesis failed [43, 74], while others revise the specification through mining environment assumptions until a robot controller is synthesized [3, 57]. In this work, the *Environment Characterization (EC)* approach automatically rewrites the specification online to incorporate any newly observed environment behaviors, as opposed to the offline approach in [57] or [3]. This work utilizes [74] to provide explanations when a controller cannot be synthesized.

• **Increase Robustness Before Execution:** Some modify the specification before synthesis such that the synthesized controller tolerates a user-defined number of safety assumption violations [25]. Others generate controllers that allow the robot to temporarily violate its guarantees when there are environment assumption violations [22]. Researchers also synthesize robust systems that maximize the ratio of environment failures to robot failures [8]. In [89] (Chapter 3), we suggest changes to the specification before execution through an analysis of low-level controllers, such that we prevent potential conflicts during execution. Here, the *Recovery* approach modifies the synthesis algorithm [10] to add "recovery" transitions into the controller. The robot can take these transitions as a fallback when a transient anomaly occurs, but the robot guarantees are never violated as in [22]. The approach is automatic and does not require users to process information and make changes as in [89].

• **Online Monitoring:** Bloem et al. synthesize a "safety shield" that monitors and cor-

rects erroneous outputs of the critical properties during execution [11]. Jones et al. use unsupervised learning to create a formula that describes normal system behaviors before execution [37]; during execution, they monitor this formula to detect anomalies. In this work, the *Integrative Negotiation* approach does not modify the outputs as in [11], but we modify the environment robot's behavior through an exchange of specifications.

• **Irregular Online Events:** Some researchers refine their controllers iteratively through grammatical inference when executing the controller in an unknown adversarial environment [28]. Some use automata learning methods to develop a control strategy for a robot with unknown dynamics [15]. For changes in topological constraints during execution, some decompose the workspace again online with new constraints [49]; some patch local strategies on top of the original strategy [60,61], while others consider alternative task plans online [20] or updates of the task representation triggered by low-level motion planners [26]. The *EC* approach in this work tackles not only changes of topological constraints as in [60] but also constraints such as dependencies among environment events and robot-induced environment events. The approach can be triggered by not only low-level motion planners but also internal or external sensors. The *Integrative Negotiation* approach in this work allows decentralized robot control with each robot having different goals, while [26] considers centralized robot control with robots sharing a common goal.

4.3 Preliminaries

In this section, the subscript r on a function, set, or formula symbol, denotes a connection to robot r.

Definition 4.3.1. Atomic Propositions To define a task for robot r, first the temporal evolution of the robot and environment behaviors of interest are abstracted by a set of atomic propositions $AP_r = \mathcal{Y}_r \cup \mathcal{X}_r$. The propositions represent the motion and actions of robot r ($\mathcal{Y}_r$ and $\mathcal{X}_r$) and other environment events including environment robots ($\mathcal{X}_r$). All propositions considered in this work are Boolean.

The set of robot/system propositions is $\mathcal{Y}_r = Reg_r \cup Act_r \cup Mem_r$. Let Reg be the set consisting of all the regions loc_i obtained from a user-defined partitioning of the robot workspace. A proposition $\pi_{loc_i,r} \in Reg_r$ is true if and only if robot r is currently heading to region loc_i. All propositions $\pi_{loc_i,r}$ in Reg_r are mutually exclusive. Act_r represents the set of actions $a_{i,r}$ that can be performed by robot r during its mission. A proposition $\pi_{a_i,r} \in Act_r$ is true if and only if robot r is initiating action $a_{i,r}$ at the moment. An action $a_{i,r}$ may be completed instantaneously or it may take finite time to complete. Note that action $a_{i,r}$ may not have completed after $\pi_{a_{i,r}}$ is set back to false and the completion of action $a_{i,r}$ is captured by sensor propositions in $\mathcal{X}_r$. Mem_r consists of 'memory' propositions that keep track of the occurrence of some robot or environment behaviors. In Example 10, the robot can conduct two actions: *pickup* and *cutBlock* (we omit '$,r$' for clarity) therefore, $Act_r = \{\pi_{pickup}, \pi_{cutBlock}\}$ and there are no memory propositions $Mem_r = \emptyset$.

The set of environment/sensor propositions $\mathcal{X}_r = \{Reg_r^c \cup Act_r^c \cup Sen_r\} \cup eAP_r \cup Sen_G$ is abstracted from the robot's sensors. Reg_r^c is the same size as $Reg_r \subseteq \mathcal{Y}_r$, and each proposition $\pi_{loc_i,r}^c \in Reg_r^c$ is true if and only if robot r is currently located in region loc_i. As before, all the propositions $\pi_{loc_i,r}^c$ in Reg_r^c are mutually exclusive. Similarly, Act_r^c contains the same number of propositions as Act_r, and a proposition $\pi_{a_i,r}^c \in Act_r^c$ is true if and

only if robot r has completed action $a_{i,r}$. Sen_r consists of propositions representing the environment as captured by the sensors of robot r. The set of environment robots' Atomic Propositions, eAP_r, consists of status information shared by the other robots operating in the same workspace. It can range from location and action status to sensor information. Sen_G is the set of global sensors accessible by all robots working in the same workspace. In Example 10, $Sen_r = \{\pi_{pickupTask}, \pi_{holdingBlock}, \pi_{blockPresent}\}$. If robot r is operating in the same workspace as robot *Bob* and *Bob* is in charge of placing a block, then we have $eAP_r = \{\pi_{placeBlock,Bob}\}$.

We use "robot" and "system" interchangeably in the following sections. Table 4.1 gives a summary of the proposition notations used in this work.

Table 4.1: Propositions Classification and Notation

Atomic Propositions AP_r	Subsets	Proposition notation	Meaning if proposition is true
System Propositions $\mathcal{Y}_r$	Act_r	$\pi_{a_i,r}$	Robot r is performing action a_i.
	Reg_r	$\pi_{loc_i,r}$	Robot r is moving towards region loc_i. At any time, there is exactly one proposition true in Reg_r.
	Mem_r	$\pi_{mem_i,r}$	For robot r, event mem_i has occurred.
Environment Propositions $\mathcal{X}_r$	Act_r^c	$\pi^c_{a_i,r}$	Robot r has completed action a_i.
	Reg_r^c	$\pi^c_{loc_i,r}$	Robot r is currently in loc_i. At any time, there is exactly one proposition true in Reg_r^c.
	Sen_r	$\pi_{sen_i,r}$	Robot r is sensing sen_i.
	eAP_r	$\pi^r_{prop,r'}$	To robot r, the environment robot r' is actuating/sensing $prop$.
	$Sen_{\mathcal{G}}$	$\pi^{\mathcal{G}}_{sen_i}$	All robots sense sen_i.

Definition 4.3.2. Robot Controller Given a set AP_r, a controller is a tuple $\mathcal{A}_r$ =

$(S_r, s_{0,r}, \mathcal{X}_r, \mathcal{Y}_r, \delta_r, \gamma_r^s, \gamma_r^e)$ where S_r is a finite set of states, and $s_{0,r} \in S_r$ is the initial state. $\mathcal{X}_r$ is the input alphabet, which is the set of environment propositions as in Def. 4.3.1. $\mathcal{Y}_r$ is the output alphabet, which is the set of robot propositions as in Def. 4.3.1. $\delta_r : S_r \times \mathcal{X}_r \to S_r$ is a transition function that given a current state and an input symbol outputs a next state. $\gamma_r^s : S_r \to 2^{\mathcal{Y}_r}$ is an output labeling function that maps a state $s_{i,r}$ to the set of robot propositions true in that state. $\gamma_r^e : S_r \to 2^{\mathcal{X}_r}$ is an input labeling function that maps a state $s_{i,r}$ to the set of environment propositions $x_{i,r}$ true for the transition to that state.

In this work, we execute hybrid controllers on the robots: each hybrid controller consists of a discrete high-level controller $\mathcal{A}_r$ and a set of continuous low-level controllers, each corresponding to a proposition $\pi \in AP_r$. These low-level controllers are in charge of carrying out the actions or sensing depending on the valuation of π.

Definition 4.3.3. Linear Temporal Logic (LTL) In this work, we automatically synthesize a robot controller $\mathcal{A}_r$ (Def. 4.3.2) from a specification written in Linear Temporal Logic (LTL). LTL formulas are constructed from a set of atomic propositions AP_r using the following grammar:

$$\varphi ::= \textbf{true} \mid \pi \in AP_r \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi.$$

We derive conjunction ($\wedge$), bi-implication ($\leftrightarrow$) and implication ($\rightarrow$), from negation ($\neg$) and disjunction ($\vee$). The temporal operators "finally" ($\Diamond$) and "always" ($\Box$) are derived from "next" ($\bigcirc$) and "until" ($\mathcal{U}$) as $\Diamond\, \varphi = \textbf{true}\, \mathcal{U} \varphi$ and $\Box\, \varphi = \neg \Diamond \neg \varphi$.

An LTL formula φ is evaluated over *traces* $\beta_r = (x_{0,r}, y_{0,r})(x_{1,r}, y_{1,r}) \ldots \in (2_r^{\mathcal{X}} \times 2_r^{\mathcal{Y}})^\omega$ of $\mathcal{A}_r$ and their corresponding runs $\sigma_r = s_{0,r}\, s_{1,r}\, s_{2,r} \ldots$. A run/trace combination can be produced by $\mathcal{A}_r$ if for every $i \in \mathbb{N}$, $s_{i+1,r} = \delta_r(s_{i,r}, x_{i+1,r})$, $x_{i,r} = \gamma_r^e(s_{i,r})$, and $y_{i,r} = \gamma_r^s(s_{i,r})$.

Whether a trace β_r satisfies an LTL formula φ at some position $i \in \mathbb{N}$ is recursively defined over the syntactic structure of the LTL formula. For proposition $p \in AP_r$, $\sigma_{r,i} \models p$ iff (if and only if) $p \in x_{i,r} \cup y_{i,r}$. For the LTL formula $\bigcirc\varphi$, $\sigma_{r,i} \models \bigcirc\varphi$ iff $\sigma_{r,i+1} \models \varphi$. A run σ_r satisfies $\varphi \mathcal{U} \varphi'$ if there exists some $i \in \mathbb{N}$ such that $\sigma_{r,i} \models \varphi'$ and for all $0 \leq j < i$, $\sigma_{r,j} \models \varphi$. A trace satisfies an LTL formula if it satisfies the formula at position 0. A run of $\mathcal{A}_r$ satisfies an LTL formula if the corresponding trace satisfies it. An LTL expression built using only propositions, negated propositions, conjunction, and next-time operators is called a "*simple LTL conjunction*". For the full semantics of LTL, the reader is referred to [18].

Definition 4.3.4. Mission Specification In this work, we restrict the class of LTL formulas considered to the Generalized Reactivity (1) (GR(1)) formulas [10]. With GR(1) specifications and the algorithm in [10], we can synthesize a controller at a lower computational complexity than with the full LTL. For the full LTL, the synthesis complexity is double exponential in the length of the formula. In comparison, the synthesis algorithm from [10] runs in time polynomial to the number of states, while the state space is exponential to the number of propositions. In the GR(1) fragment a mission specification φ_r is:

$$\begin{aligned} \varphi_r &= \varphi_{e,r} \rightarrow \varphi_{s,r} \\ &= (\varphi^i_{e,r} \wedge \varphi^t_{e,r} \wedge \varphi^g_{e,r}) \rightarrow (\varphi^i_{s,r} \wedge \varphi^t_{s,r} \wedge \varphi^g_{s,r}). \end{aligned} \tag{4.1}$$

The subformulas of φ_r with superscript 'i' stand for initial conditions, with superscript 't' for transitions, and with superscript 'g' for goals. The 'i' is overloaded here as an abbreviation, but should not be confused with an index variable i in the rest of the paper.

$\varphi^i_{e,r}$ is the initial condition of the environment, in the form of a Boolean formula over

$\mathcal{X}_r$. $\varphi^i_{s,r}$ is the initial condition of the robot, in the form of a Boolean formula over $\mathcal{X}_r \cup \mathcal{Y}_r$.

$\varphi^t_{e,r}$=$\bigwedge \square \psi^t_{e,r}$ are the environment safety assumptions and $\psi^t_{e,r}$ is a Boolean combination of terms from $\mathcal{X}_r \cup \mathcal{Y}_r \cup \bigcirc\mathcal{X}_r$, where $\bigcirc V$=$\{\bigcirc v \mid \forall v \in V\}$. The formula $\varphi^t_{e,r}$ denotes the assumptions about the environment behaviors during execution.

$\varphi^t_{s,r} = \bigwedge \square \psi^t_{s,r}$ is the system safety guarantees, with $\psi^t_{s,r}$ being a Boolean combination of terms from $\mathcal{X}_r \cup \mathcal{Y}_r \cup \bigcirc \mathcal{X}_r \cup \bigcirc \mathcal{Y}_r$. These are user-provided constraints over the behavior of the robot that must be maintained at all times.

$\varphi^g_{e,r} = \bigwedge^p_{i=1} \square \diamond \psi^{g,i}_{e,r}$ are the environment liveness properties, where each $\psi^{g,i}_{e,r}$ is a Boolean formula over $\mathcal{X}_r \cup \mathcal{Y}_r$. These represent environment states that are assumed to be reached repeatedly during execution.

$\varphi^g_{s,r} = \bigwedge^q_{j=1} \square \diamond \psi^{g,j}_{s,r}$ are the system liveness properties, i.e. the robot goals, where each $\psi^{g,j}_{s,r}$ is a Boolean formula over $\mathcal{X}_r \cup \mathcal{Y}_r$. These are conditions that should be satisfied by the robot repeatedly during execution.

The specifications for Example 10 are shown in Spec. 4. The initial conditions are that there are no blocks and the robot is idle; it has not received a pickup task yet (line 1-2). The environment safety assumption (line 3) is that a block is present when the robot receives a pickup command. The system safety guarantees (line 4-5) are that the robot picks up the block when it receives a command and it cuts the block when it senses a block in its manipulator. If the block is placed by another robot, we replace line 2 and 3 in Spec. 4 with Spec. 5. We omit topological constraints in the specifications throughout the paper for clarity.

Specification 4 Block Cutting Example

$$
\begin{aligned}
1\ \varphi^{i}_{s,r} &= \neg\pi_{pickup} \wedge \neg\pi_{cutBlock} \\
2\ \varphi^{i}_{e,r} &= \neg\pi_{pickupTask} \wedge \neg\pi_{holdingBlock} \wedge \neg\pi_{blockPresent} \\
3\ \varphi^{t}_{e,r} &= \square((\bigcirc\pi_{pickupTask} \wedge \neg\bigcirc\pi_{holdingBlock}) \rightarrow (\bigcirc\pi_{blockPresent}) \\
4\ \varphi^{t}_{s,r} &= \square((\bigcirc\pi_{pickupTask} \wedge \bigcirc\pi_{blockPresent}) \leftrightarrow \bigcirc\pi_{pickup}) \wedge \\
5 &\qquad \square(\pi_{holdingBlock} \leftrightarrow \bigcirc\pi_{cutBlock})
\end{aligned}
$$

Specification 5 Block Cutting Example - Two Robots

$$
\begin{aligned}
2\ \varphi^{i}_{e,r} &= \neg\pi_{pickupTask} \wedge \neg\pi_{holdingBlock} \wedge \neg\pi_{blockPresent} \wedge \neg\pi_{placeBlock,Bob} \\
3\ \varphi^{t}_{e,r} &= \square(\neg\bigcirc\pi_{blockPresent} \rightarrow \bigcirc\pi_{placeBlock,Bob})
\end{aligned}
$$

A specification φ_r is realizable if a controller $\mathcal{A}_r$ satisfying φ_r (on all of its traces) can be successfully synthesized; the specification φ_r is unrealizable otherwise.

Ideally, the user wants the robot to maintain the properties in $\varphi^{t}_{s,r}$ and complete the goals specified in the formulas $\varphi^{g}_{s,r}$ when executing the controller $\mathcal{A}_r$. However, from Eq. 4.1, if $\varphi_{e,r}$ is false in a trace of the automaton $\mathcal{A}_r$, then the trace can satisfy Eq. 4.1 even when $\varphi_{s,r}$ is false due to the implication structure, therefore robot behaviors may not achieve the system specification $\varphi_{s,r}$. This work leverages [23] where possible robot behaviors that actively make the environment assumptions false are prohibited.

Definition 4.3.5. Game Structures Each robot controller $\mathcal{A}_r$ (Def. 4.3.2) is constructed by finding a winning strategy for the robot in a two-player game between the environment and the robot [10]. The game $\mathcal{G}_r = (\mathcal{X}_r, \mathcal{Y}_r, \Theta_r, \rho^{e}_{r}, \rho^{s}_{r}, \phi_r)$ has the following components:

$\mathcal{X}_r$ and $\mathcal{Y}_r$ are as described in Def. 4.3.1. A position of the game is an assignment to the variables in $\mathcal{X}_r \cup \mathcal{Y}_r$. $\Theta_r \subseteq 2^{(\mathcal{X}_r \cup \mathcal{Y}_r)}$ is the set of initial positions. The input constraints

$\rho_r^e \subseteq 2^{(\mathcal{X}_r \cup \mathcal{Y}_r)} \times 2^{\mathcal{X}_r}$ define the possible incoming inputs, i.e. possible environment behaviors, based on the current position. $\rho_r^s \subseteq 2^{(\mathcal{X}_r \cup \mathcal{Y}_r)} \times 2^{(\mathcal{X}_r \cup \mathcal{Y}_r)}$ defines the next possible positions based on the current position. The winning condition of the robot ϕ_r is given in the form of an LTL formula:

$$(\square \diamondsuit \psi_{e,r}^{g,1} \wedge \cdots \wedge \square \diamondsuit \psi_{e,r}^{g,p}) \rightarrow (\square \diamondsuit \psi_{s,r}^{g,1} \wedge \cdots \wedge \square \diamondsuit \psi_{s,r}^{g,q}),$$

with each $\psi_{e,r}^{g,p}$ or $\psi_{s,r}^{g,q}$ being a Boolean formula from $\varphi_{e,r}^g$ and $\varphi_{s,r}^g$ respectively.

4.4 Problem Formulation

We say an environment robot is *collaborative* if it has its own specification $\varphi_{r_{env}}$, can communicate and can attempt to adjust its own behavior following requests from robot r. An *unresponsive* environment robot is one that cannot be communicated with or that cannot adjust its own behaviors following requests from robot r; it is treated as any other uncontrollable environment element.

As described in Section 4.3, we first abstract the robot and environment behaviors to propositions. With these propositions, a user writes a mission specification $\varphi_r = \varphi_{e,r} \rightarrow \varphi_{s,r}$ and synthesizes a provably-correct controller $\mathcal{A}_r$ with [10]. This specification may include explicit assumptions about environment behaviors, captured in $\varphi_{e,r}$. During execution, when robot r is in a state $s_{i,r}$ of the controller $\mathcal{A}_r$, the next environment state $x_{i+1,r}$ may be different from the user expectation and violate the safety assumptions in $\varphi_{e,r}^t$: $(s_{i,r}, x_{i+1,r}) \not\models \varphi_{e,r}^t$. Note that one cannot detect liveness assumption violations in finite executions, therefore we focus on safety violations.

In such cases, due to the game-based synthesis formulation, a controller synthesized with [10] will not have a valid next state, which practically means the robot will throw an exception and fail. In this paper we create a layered approach to deal with assumption violations, solving the following problem:

Problem 3. Given a specification φ_r and a controller $\mathcal{A}_r$ synthesized from φ_r, construct a controller $\mathcal{A}'_r$ based on φ_r that increases robustness against violations of environment safety assumptions $\varphi^t_{e,r}$ during execution, i.e., such that the controller $\mathcal{A}'_r$ satisfies the specification φ_r, including on all traces β_r starting from a state $s_{r,i+1}$ reached immediately after an environment assumption violation and for which there exists a winning strategy from some state in β_r after the final assumption violation.

This work considers environment assumption violations. We assume sensors provide correct information about the environment and actuators are able to perform actions correctly. Dealing with sensor and actuation noise and faults is outside of the scope of this paper. We will tackle Problem 3 with a particular emphasis on multi-robot scenarios, where the controller operates in a partially collaborative environment.

4.5 Overview

To address Problem 3, consider the following two different scenarios where robot r's assumptions may be violated when executing a provably-correct robot controller $\mathcal{A}_r$:

S.1 robot r is operating in a workspace by itself or with unresponsive robots;

S.2 robot r is working in a workspace shared with some other collaborative robots.

In scenario S.1, if one of the environment safety assumptions $\varphi^t_{e,r}$ is violated during execution, then it must be due to some incorrectly modeled environment behaviors in the specification φ_r. Since the assumptions $\varphi^t_{e,r}$ are falsified, the controller $\mathcal{A}_r$ does not have a next state. To restore the desired robot behavior $\varphi_{s,r}$, we proposed two different approaches in [86].

The first is an offline approach called *Recovery*. It introduced a modified controller synthesis algorithm based on [10]. This synthesis algorithm adds extra "recovery" transitions; the robot will continue and complete its task with this "recovery" controller as long as the violations of $\varphi^t_{e,r}$ are transient and do not cause the violation of $\varphi^t_{s,r}$. In this paper (Section 4.6), the approach is improved to include states and transitions that do not guarantee the satisfaction of robot goals $\varphi^g_{s,r}$ but the robot guarantees $\varphi^t_{s,r}$ hold: the robot can take these "safe" moves when no better moves are available. This guarantees that the robot controller $\mathcal{A}_r$ has a next state, as opposed to [86], when the robot can safely remain in these "safe" non-winning states. Once the violations of the environment assumptions $\varphi^t_{e,r}$ disappear, the robot can proceed to its goals. The approach solves Problem 3 for transient assumption violations by constructing a controller $\mathcal{A}'_r$ that (1) satisfies, whenever possible, the robot guarantees $\varphi^t_{s,r}$ when environment safety assumptions $\varphi^t_{e,r}$ are violated and (2) satisfies the full robot specification $\varphi_{s,r}$ when the violations end and the robot is eventually in a winning state, i.e., a state from which the satisfaction of the specification φ_r can be ensured.

The second approach in [86], *Environment Characterization*, is applied during execution. The approach automatically relaxes the environment safety assumptions $\varphi^t_{e,r}$ during execu-

tion, when the robot observes any incorrectly modeled environment behaviors. In this paper (Section 4.7), the approach has been changed to account for the most-recently-observed environment behaviors as opposed to the previous observation in [86]. The approach here ensures that any new environment information included in the specification is up-to-date. The approach solves Problem 3 by automatically relaxing the environment assumptions and synthesizing a controller $\mathcal{A}'_r$ that satisfies the guarantees $\varphi_{s,r}$ for a larger set of possible environment behaviors, including those observed during execution that violated the original assumptions.

If robot r is sharing its workspace with other collaborative robots as in scenario S.2, then violations of the environment safety assumptions $\varphi^t_{e,r}$ may be caused by unexpected behaviors performed by the environment robots. For such cases, we proposed the *Integrative Negotiation* approach to resolve the conflict [88]. The approach attempts to modify the environment behaviors by changing the behavior of one of the environment robots. Partial specifications are exchanged between the pair of robots in conflict. Incorporating the plan of the other robot, if possible, enables the robots to continue their tasks. In this paper (Section 4.8), the approach now includes a new step where the two robots in negotiation both modify their specifications, such that if successful, no further assumption violations will be caused by the other robot. In [88], it is always possible that the robot not incorporating the other's specification violates the other's assumptions again. The approach solves Problem 3 by automatically updating a specification φ'_r with the partial specification of the environment robot. The controller $\mathcal{A}'_r$ synthesized from φ'_r ensures that each robot satisfies the assumptions the other robot is making regarding the robot's behavior.

The three proposed approaches together create a system that can handle unexpected events when a robot is operating in a shared environment. Controllers created with the *Recovery* approach are more forgiving of any violations, but the robot can only resume making progress towards the completion of its task if the violation eventually disappears. If the violation persists and the robot cannot make progress towards its goal, the *Environment Characterization* approach in Section 4.7 can re-enable progress. For these two approaches, the robot is passive: it only checks if it can continue its task with guarantees when unexpected events arise. If the robot is operating in a shared workspace with some collaborative robots, the robot can play an active role and ask the environment robot to modify its behaviors with the *Integrative Negotiation* approach, which changes the robot environment behaviors. The system solves Problem 3 by increasing robustness of the provably-correct controller execution with the three approaches.

If the three approaches cannot create a new controller $\mathcal{A}_r'$, i.e. the specification is unrealizable, feedback regarding the unrealizability of the specification φ_r' is provided to the user leveraging the techniques in [74]. Other approaches (e.g. [43, 57]) can be used as well. The user can modify the specification φ_r' based on the feedback returned.

All the tasks considered in this work are encoded using the activation-completion paradigm described in [75]. Compared with the examples in [86] and [88], this results in a more realistic modeling of real-world scenarios where actions take time to complete and can be aborted before they are completed.

Fig. 4.1 gives a summary of how the three approaches connect with each other in this system. Table 4.2 summarizes the superscript notations used in the following sections.

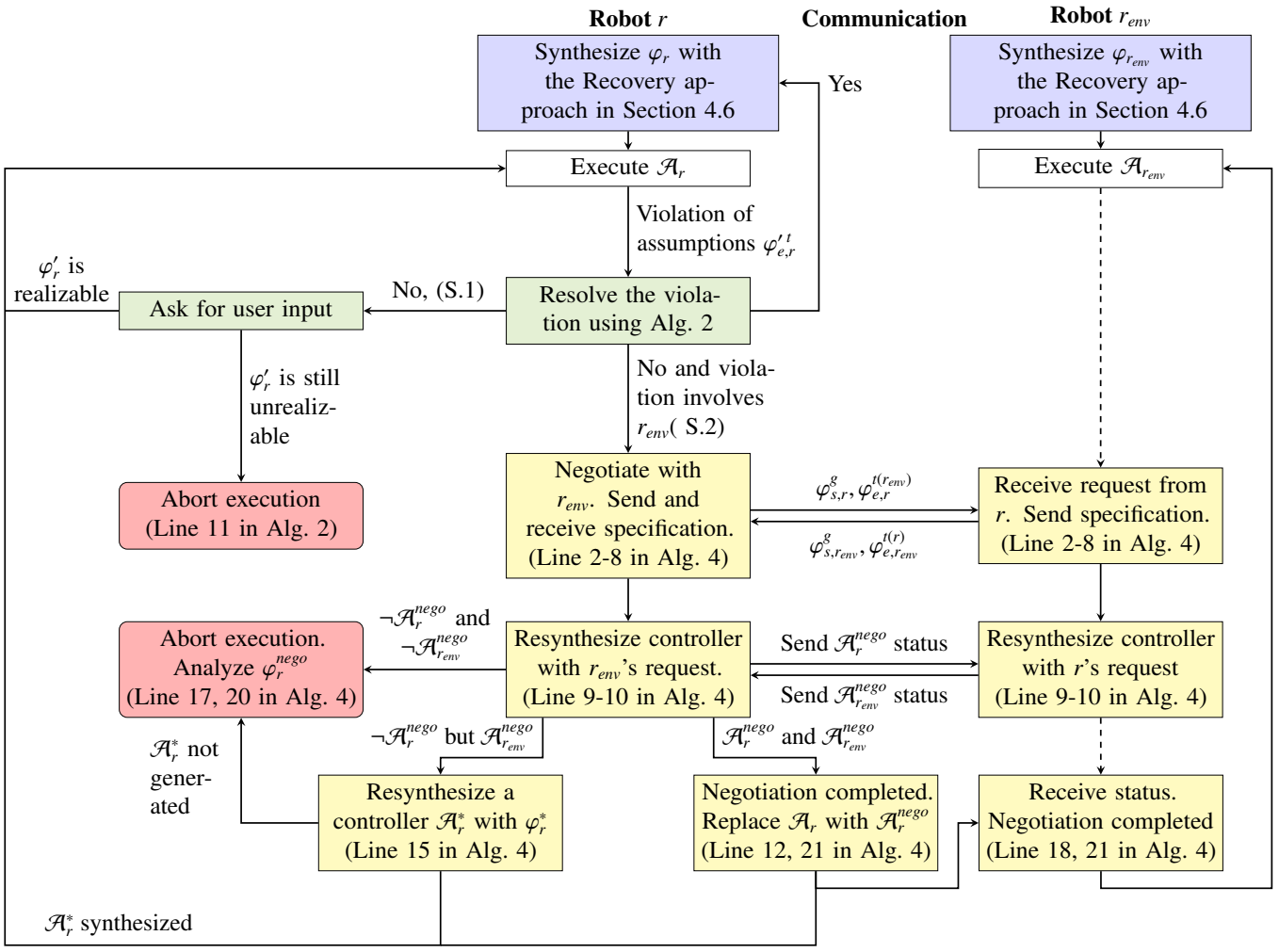


Figure 4.1: Connection of the three approaches. The purple boxes highlight the Recovery approach in Section 4.6; the green boxes highlight the Environment Characterization approach in Section 4.7; the yellow boxes describe the Integrative Negotiation approach in Section 4.8; finally, we abort the execution if we reach the red boxes.

Table 4.2: Explanation of the Superscript Notations for Specification φ_r and Controller $\mathcal{A}_r$

Section	Type	Superscript Notation	Example	Meaning of the superscript
4.7	Environment Characterization Result	h'	$\mathcal{A}'_r$, $\varphi'^{\,i}_{e,r}$	Modified h via the Environment Characterization Approach. When the execution just begins, h' is the same as h.
	Extra simple LTL conjunction	$h^{\overset{\text{conj}}{\text{EC1}}}$	$\varphi^{t,\overset{\text{conj}}{\text{EC1}}}_{e,r}$	A simple LTL conjunction created from the Environment Characterization approach.
	Specification updated with simple LTL conjunction	h^{EC1}	$\varphi^{t,EC1}_{e,r}$	h updated with the simple LTL conjunctions $h^{\overset{\text{conj}}{\text{EC1}}}$ found with the Environment Characterization approach.
4.8	Integrative Negotiation Result	h^*	$\mathcal{A}^*_r$, $\varphi^{*\,i}_{e,r}$	Modified h via the Integrative Negotiation Approach. When the execution just begins, h^* is the same as h.
	Snippets to robot r_{env}	$h^{(r_{env})}$	$\varphi^{t(r_{env})}_{e,r}$	Environment Assumptions to send to the environment robot r_{env}
	Intermediate Integrative Negotiation Result	h^{nego}	$\mathcal{A}^{nego}_r$, $\varphi^{t\,nego}_{e,r}$	Intermediate h inside the Integrative Negotiation approach.

4.6 Recovery

The synthesis problem for Generalized Reactivity (1) specifications [10] is typically reduced to finding a winning strategy for the robot, or the system player, in a suitable game structure (Definition 4.3.5). We call the positions of a game structure from which the system player can win the game the *winning positions*. These positions can be found by evaluating the following equation:

$$W = \nu Z.\bigcap_{j=1}^{q} \mu Y.\bigcup_{i=1}^{p} \nu X.\diamondsuit((\phi_j^s \cap Z) \cup Y \cup (\overline{\phi}_i^e \cap X)). \tag{4.2}$$

In Eq. 4.2, ν is the *greatest fixpoint operator* and μ is the *least fixpoint operator*. Given a monotone function $f : 2^J \to 2^J$ for some set J, $\mu X.f(X)$ denotes the limit of $f(f(\ldots(f(\emptyset))\ldots))$, whereas $\nu X.f(X)$ represents the limit of $f(f(\ldots(f(2^J))\ldots))$. The $\diamondsuit$ operator takes a set of states $T \subseteq 2^{(\mathcal{X}_r \cup \mathcal{Y}_r)}$ and computes the positions that the system player can take a transition to T within one step. Formally:

$$\begin{aligned}\diamondsuit(T) = \{(x,y) \in 2^{\mathcal{X}_r \cup \mathcal{Y}_r} \mid \forall x' \subseteq \mathcal{X}_r.\exists y' \subseteq \mathcal{Y}_r.\\ (x,y,x') \in \rho_r^e \to (x,y,x',y') \in \rho_r^s \wedge (x',y') \in T\}.\end{aligned}$$

Note that $\diamondsuit$ is a monotonic operator as T occurs only positively in the definition of the operator, i.e., adding elements to a set with the operator applied can only make the resulting position set larger. Also, in the fixpoint equation (Eq. 4.2), ϕ_j^s represents the positions satisfying the jth system goal $\psi_{s,r}^{g,j}$, whereas $\overline{\phi}_i^e$ represents the positions that *do not* satisfy the ith environment goal $\psi_{e,r}^{g,i}$ (for all $1 \leq i \leq p$ and $1 \leq j \leq q$).

The outermost fixpoint of the equation defining W successively approximates the positions that are winning for the system player. In every iteration of the outermost fixpoint evaluation, it is checked that each system goal can be reached (if the environment reaches its goals infinite often). This idea is represented by the $\bigcap_{j=1}^{q}$ operator that iterates over the system goals. The middle least fixpoint operator then successively approximates the positions that can reach a system goal $\psi_{s,r}^{g,j}$, starting with the positions that are closest to the goal and ending with the positions that are farthest away from the goal.

The $\bigcup_{i=1}^{p}$ operator iterates over all environment goals; the strategy may wait for any of them to be reached on the way to the next system goal. The inner-most greatest fixpoint then implements the allowance to wait: we want to find the greatest set of positions for which leaving the set implies that an environment goal has been reached.

The $\diamondsuit(\ldots)$ part in Eq. 4.2 finally implements the aim of the system player to always take transitions such that either:

T.1 a system goal is reached and the position is still winning afterwards (so that the execution of the strategy can continue in a provably-correct fashion),

T.2 the next position is closer to the current system goal, or

T.3 the strategy is currently waiting for the next environment goal to be reached.

These transitions are exactly the ones given to the $\diamondsuit$ operator. The $\diamondsuit$ operator then checks and returns positions that fulfill T.1, T.2 or T.3 by playing transitions in ρ_r^s, provided that the environment only picks transitions in ρ_r^e.

A strategy for the system player to win the game can then be extracted by cycling through the system goals and always picking transitions that:

1. were considered in the evaluation of the $\textcircled{\diamond}(\ldots)$ operator as early in the middle least fixpoint μY evaluation as possible,
2. while only considering fully evaluated greatest fixpoints νZ and νX.

This ensures that the strategy never moves away from the (next) system goal and can only get closer to it. This definition of the winning strategy naturally leads to only including transitions $(x, y, x', y') \in (2^{\mathcal{X}_r \cup \mathcal{Y}_r})^2$ for which $(x, y) \in W$ and $(x, y, x') \in \rho_r^e$. Hence, the extracted strategy is not robust to transient violations of the environment safety assumptions, as the implementation does not contain any transitions that witness the violation of environment safety assumptions.

In order to allow the generated implementation to *recover* from transient violations of the environment assumptions, we modify the strategy extraction procedure; we add transitions that satisfy ρ_r^s but do not satisfy ρ_r^e. These transitions preserve the safety of the robot, and we first add transitions that lead to winning states, as in [86], and then transitions that do not lead to winning states, as described below.

4.6.1 Adding Recovery Transitions to Winning States

After evaluating W according to the equation above, for every $(x, y) \in 2^{\mathcal{X}_r \cup \mathcal{Y}_r}$ and $x' \in 2^{\mathcal{X}_r}$, we check if there is *some* $y' \in 2^{\mathcal{Y}_r}$ for which $(x, y, x', y') \in \rho_r^s$ and for which

$(x', y') \in W$. Whenever this is the case, we pick a value of y' such that (x', y') appears as early as possible in the least fixpoint evaluation for the next system goal, that is, the robot is closest to satisfying the current goal.

While this change to the strategy extraction routine of the synthesis algorithm is relatively small, it has a huge effect: instead of considering only transitions that satisfy the environment assumptions, the newly generated implementation or strategy supports all incoming environment player moves that do not violate the specification in the long run.

If environment assumption violations are not transient, the implementation **may** enter a livelock. This is the case if the only valid response of the system at a point in time is moving further away from the current system goal. However, as we prefer transitions that do not do so, this only occurs in cases where moving away from the goal is unavoidable. Whenever possible, the generated controller ignores transient environment assumption violations, making it robust to irrelevant deviations from the environment assumptions.

4.6.2 Adding Recovery Transitions to Non-Winning States

In some cases, the controller cannot reach a winning state following an environment assumption violation: the controller cannot ensure the specification will hold along its execution as some existing environment behaviors will prevent that. Yet, this environment behavior may not occur forever. Thus, we add alternate transitions (that satisfy ρ_r^s) to positions that are not in W to the controller. These positions are such that W is reachable by some sequence of transitions that are all in ρ_r^s. By preferring transitions that move closer

to positions in W, the generated implementation can still recover as quickly as possible in such a case. Adding system behaviors for cases in which the controller can no longer satisfy the specification without the help of the environment has been explored in [9]. However, they did not consider adding such transitions only if these positions actually allow recovery towards positions from which the specification can be enforced to hold again.

Algorithm 1 Strategy extraction algorithm. The game structure is given as a tuple $\mathcal{G} = (\mathcal{X}_r, \mathcal{Y}_r, \Theta_r, \rho_r^e, \rho_r^s, \phi_r)$, the family of results $\{P_{j,c,i}\}_{1 \leq c < cmax, i < imax}$ is the set of intermediate results of evaluating Equation 4.2, where c is the iteration of the μY fixpoint and i and j are the indices of the environment liveness assumptions and system goals, as in Equation 4.2. The variable *ToDo* stores the states whose successors still have to be computed as a set, so duplicates are removed. We assume that the number of system goals q is ≥ 1 and for all $1 \leq j \leq q$, ϕ_j^s represents the states listed in the jth system goal. All indices in **for** loops are iterated through in ascending order.

procedure ExtractImplementation
  $S_{0,r} \leftarrow (\bigcup_{1 \leq c < cmax, i < imax} P_{j,c,i}) \times \{1\}$
  $ToDo \leftarrow S_{0,r}$, $S_r \leftarrow \emptyset$, $\delta_r = \emptyset$, $\gamma_r^s \leftarrow \emptyset$, $\gamma_r^e \leftarrow \emptyset$
  $B_{j,c} \leftarrow \mu^c Y.\langle\!\langle((\phi_j^s \cap Z) \cup Y)$ for all $c \in \mathbb{N}$ and $1 \leq j \leq q$
5: **while** $ToDo \neq \emptyset$ **do**
    $(s, j) \leftarrow \mathrm{popElement}(ToDo)$, $S_r \leftarrow S_r \cup \{(s, j)\}$
    $\gamma_r^s \leftarrow \gamma_r^s \cup \{(s, j) \mapsto s|_{\mathcal{Y}_r}\}$, $\gamma_r^e \leftarrow \gamma_r^e \cup \{(s, j) \mapsto s|_{\mathcal{X}_r}\}$
    for $X' \subseteq \mathcal{X}_r$ **do**
      for $c < cmax_j$ **do**
10: **for** $i < imax$ **do**
          if $\exists Y' \subseteq \mathcal{Y}_r : (X', Y') \in P_{j,c,i}$
                $\wedge\, (s, X', Y') \models \rho_r^s$ **then**
            $Y' \leftarrow$ some $Y' \subseteq \mathcal{Y}$ s.t. $(X', Y') \in P_{j,c,i} \wedge (s, X', Y') \models \rho_r^s$
            if $(X', Y') \models \phi_j^s$ **then** $j' \leftarrow (j \mod n) + 1$
            else $j' \leftarrow j$
15: $\delta_r \leftarrow \delta_r \cup \{((s, j), X') \mapsto ((X', Y'), j')\}$
            if $((X', Y'), j') \notin S_r$ **then**
              $ToDo \leftarrow ToDo \cup \{((X', Y'), j')\}$
            break to line 8
      for $c \in \mathbb{N}$ such that $c = 0$ or $B_{j,c} \neq B_{j,c-1}$ **do**
20: **if** $\exists Y' \subseteq \mathcal{Y}_r : (X', Y') \in B_{j,c}$ **then**
          $Y' \leftarrow$ some $Y' \subseteq \mathcal{Y}$ s.t. $(X', Y') \in B_{j,c} \wedge (s, X', Y') \models \rho_r^s$
          $\delta_r \leftarrow \delta_r \cup \{((s, j), X') \mapsto ((X', Y'), j)\}$
          if $((X', Y'), j) \notin S_r$ **then**
            $ToDo \leftarrow ToDo \cup \{((X', Y'), j)\}$
25: **break** to line 8
  return $(\mathcal{X}_r, \mathcal{Y}_r, S_r, S_{r,0}, \delta_{r,s}, \gamma_r^e, \gamma_r^s)$

The resulting overall strategy extraction procedure is shown in Algorithm 1. The computation of the backup behavior outside of W is given from line 19 onwards. Our Recovery approach is light-weight as it does not alter the structure of the generated implementations

and it does not modify the specification or the computation of the winning positions. This is in contrast to more sophisticated approaches to synthesis of error-resilient systems such as [25] in which the generated implementation plans ahead and computation times become much longer. As we integrate *Recovery* with *Environment Characterization* (Section 4.7) in this work, the approach in [25] is not needed here.

4.7 Environment Characterization (EC)

First introduced in [86], the Environment Characterization approach relaxes and updates the environment assumptions $\varphi^t_{e,r}$ during execution based on any newly-observed and previously-disallowed behaviors of the environment; the updated assumptions allow for more environment behaviors than originally assumed: therefore $\mathcal{A}'_r$, the controller synthesized from the updated specification if it is realizable, guarantees correct robot behavior in a larger range of environment behaviors. In this approach, φ_r is modified to φ'_r, where initially, all components are the same:

$$\varphi'_r = \varphi_r = \varphi'^{\,i}_{e,r} \wedge \varphi'^{\,t}_{e,r} \wedge \varphi'^{\,g}_{e,r} \rightarrow \varphi'^{\,i}_{s,r} \wedge \varphi^t_{s,r} \wedge \varphi^g_{s,r}.$$

Note that if users do not know anything about the environment behavior a priori, they can start with $\varphi^t_{e,r} = \square(False)$, which means the environment has no possible behaviors. During execution, our approach then adds possible environment behaviors automatically based on observed behaviors.

4.7.1 Runtime Monitoring

During execution, the approach leverages techniques from the runtime verification community [48, 55] and particularly with LTL [1], where a system includes monitors that continuously observe and verify some properties of interest for the system and its environment; the monitors report system success and failure. In this work, our approach monitors the environment safety assumptions $\varphi'^{\,t}_{e,r}$ in the specification φ'_r. At every step of the controller execution, the assumptions $\varphi'^{\,t}_{e,r}$ are checked against the current state $s_{i,r}$ of robot r and the incoming sensor values $x_{i+1,r}$. Our approach detects assumption violation and rewrites the assumptions $\varphi'^{\,t}_{e,r}$ based on the incoming sensor values $x_{i+1,r}$ to obtain a closer approximation of the actual environment behaviors, as described next.

4.7.2 Automatic rewriting of the specification

We automatically incorporate newly observed environment behaviors $x_{i+1,r}$ into the current environment safety assumptions $\varphi'^{\,t}_{e,r}$, thus including additional behaviors unaccounted for in the previous synthesized controller $\mathcal{A}'_r$.

We rewrite the assumptions using two different approximations of the allowed environment transitions: coarse and fine. The algorithm switches from the coarse to the fine only when needed due to unrealizability, as it takes longer for the finer one to capture all possible environment transitions. Formally, the assumptions are captured using two different simple LTL conjunctions as shown in Eq. 4.3 and Eq. 4.4 below.

$$\varphi_{e,r}^{t,\text{EC1}\,\text{conj}} = (\bigwedge_{\pi \in x_{i+1,r}} \bigcirc \pi \wedge \bigwedge_{\pi \in \mathcal{X}_r \setminus x_{i+1,r}} \neg \bigcirc \pi), \tag{4.3}$$

$$\varphi_{e,r}^{t,\text{EC2}\,\text{conj}} = (\bigwedge_{\pi \in \gamma_r^e(s_{i,r})} \pi \wedge \bigwedge_{\pi \in \mathcal{X}_r \setminus \gamma_r^e(s_{i,r})} \neg \pi \wedge \bigwedge_{\pi \in x_{i+1,r}} \bigcirc \pi \wedge \bigwedge_{\pi \in \mathcal{X}_r \setminus x_{i+1,r}} \neg \bigcirc \pi). \tag{4.4}$$

Eq. 4.3 appends the incoming valuation of the environment propositions $x_{i+1,r}$ to the assumptions $\varphi_{e,r}^{\prime\, t}$ as disjuncts; while Eq. 4.4 appends the evaluation of the current environment propositions $\gamma_r^e(s_{i,r})$ and that of the next environment propositions $x_{i+1,r}$ to the assumptions $\varphi_{e,r}^{\prime\, t}$ as disjuncts. The next environment propositions $x_{i+1,r}$ are a subset of $\mathcal{X}_r$, which include propositions in Sen_r, Reg_r^c and Act_r^c.

Eq. 4.3 captures more environment behaviors than Eq. 4.4. When appended to the assumptions $\varphi_{e,r}^{\prime\, t}$, the new assumptions using Eq. 4.3 allow more environment behaviors, while the assumptions $\varphi_{e,r}^{\prime\, t}$ created with Eq. 4.4 are more specific and only add environment behaviors that match a two-step sequence of current and next environment proposition values. Both Eq. 4.3 and Eq. 4.4 relax the environment behaviors; their addition to the environment safety assumptions $\varphi_{e,r}^{\prime\, t}$ accounts for more possible environment behaviors in the specification $\varphi_r^{\prime}$. For example, consider there is only one sensor $\pi_{doorClosed}$. When the execution starts, the door is open ($\pi_{doorClosed}$ is false). Then, the door is closed in the next step ($\pi_{doorClosed}$ is true). The conjunction added with Eq. 4.3 is ($\bigcirc \pi_{doorClosed}$), while the conjunction added with Eq. 4.4 is ($\neg \pi_{doorClosed} \wedge \bigcirc \pi_{doorClosed}$), which explicitly specifies a possible environment transition and does not allow as many new environment behaviors as that with Eq 4.3.

Algorithm 2 shows an update to the environment safety assumptions $\varphi_{e,r}^{\prime\, t}$ with the newly observed environment behaviors $x_{i+1,r}$. To prepare for resynthesis of the robot controller

Algorithm 2 One iteration of Environment Characterization

procedure EnvironmentCharacterization
2: $\varphi'^{\,i}_{e,r} = \bigwedge_{\pi \in x_{i+1,r}} \pi \wedge \bigwedge_{\pi \in \mathcal{X}_r \setminus x_{i+1,r}} \neg\pi$
$\varphi'^{\,i}_{s,r} = \bigwedge_{\pi \in \gamma^s_r(s_{i,r})} \pi \wedge \bigwedge_{\pi \in \mathcal{Y}_r \setminus \gamma^s_r(s_{i,r})} \neg\pi$
4: $\varphi^{t,EC1}_{e,r} \leftarrow \varphi^{t,EC1}_{e,r} \vee \varphi^{t,\mathrm{EC1}}_{e,r}$ (Eq. 4.3) $\quad$ conj
$\varphi^{t,EC2}_{e,r} \leftarrow \varphi^{t,EC2}_{e,r} \vee \varphi^{t,\mathrm{EC2}}_{e,r}$ (Eq. 4.4) $\quad$ conj
6: $\varphi'^{\,t}_{e,r} = \square(\bigwedge \psi^t_{e,r} \vee (\varphi^{t,EC1}_{e,r}))$
$\mathcal{A}'_r \leftarrow \mathrm{SYNTHESIS}(\varphi'_r)$
8: **if** φ'_r is unrealizable (No $\mathcal{A}'_r$ is generated) **then**
$\varphi'^{\,t}_{e,r} = \square(\bigwedge \psi^t_{e,r} \vee (\varphi^{t,EC2}_{e,r}))$
10: $\mathcal{A}'_r \leftarrow \mathrm{SYNTHESIS}(\varphi'_r)$
return $\mathcal{A}'_r$ if generated else return None

$\mathcal{A}'_r$, when the environment assumptions $\varphi'^{\,t}_{e,r}$ are violated, Alg. 2 first automatically updates the environment and system initial conditions $\varphi'^{\,i}_{e,r}, \varphi'^{\,i}_{s,r}$ based on the current observed state (lines 2,3) to ensure the robot restarts in a valid state.

Eq. 4.3 is attempted before Eq. 4.4, added as a disjunct to the environment safety requirements $\varphi'^{\,t}_{e,r}$ (lines 4,6). If a new controller $\mathcal{A}'_r$ is successfully synthesized from the specification φ'_r, which now includes the newly observed environment behaviors $x_{i+1,r}$ (line 7), then the new controller $\mathcal{A}'_r$ can start from the current state and reach all the goals while satisfying the system safety guarantees, provided that the new environment assumptions $\varphi'^{\,t}_{e,r}$ are not violated.

If the specification φ'_r is unrealizable after appending the simple LTL conjunctions from Eq. 4.3 (line 8), then Alg. 2 uses Eq. 4.4 to relax the specification φ'_r instead (lines 5,9). If the modified specification φ'_r is realizable with either Eq. 4.3 or Eq. 4.4 and a new robot controller $\mathcal{A}'_r$ is synthesized, then the execution resumes with the latest controller $\mathcal{A}'_r$.

Algorithm 3 Online resilience to unexpected events

```
    procedure UnexpectedEventsResilience
2:    A'_r ← EnvironmentCharacterization (Alg. 2)
      if φ'_r is still unrealizable (No A'_r is generated) then
4:      if S.1: an uncontrollable environment then
          Provide feedback with specification analysis [74]
6:        φ'^g_{e,r} = φ'^g_{e,r} ∧ User input
          A'_r ← SYNTHESIS(φ'_r)
8:      else
          A'_r, φ'_r ← IntegrativeNegotiation (Section 4.8)
10:     if φ'_r is unrealizable (No A'_r is generated) then
          Abort Execution return
12:   Continue Execution
```

Otherwise, after Eq. 4.3 was attempted, if the specification φ'_r is still unrealizable when incorporating Eq. 4.4 (line 3 in Alg. 3), we consider two possible situations: if the environment assumptions violation is caused by a collaborative robot operating in the same workspace (S.2), then the Integrative Negotiation approach in Section 4.8 is pursued (line 9 in Alg. 3); otherwise (S.1) Alg. 3 provides an analysis of the unrealizable specification φ'_r to the user using the technique from [74] (line 5 in Alg. 3).

Interpreting the feedback, the user may include in the specification φ'_r new environment livenesses (line 6 in Alg. 3), which add assumptions about behaviors that the environment will repeatedly exhibit. Intuitively, the environment livenesses $\varphi'^{g}_{e,r}$ explicitly remove some perpetual environment behaviors that are preventing the robot from reaching its goal. If the revised specification φ'_r incorporating the user inputs is realizable, then the execution continues with the new controller $\mathcal{A}'_r$. However, if adding new environment livenesses does not render the specification φ'_r realizable, then the execution is aborted. The user should then reexamine the specification φ'_r, since no provably-correct robot controller $\mathcal{A}'_r$ exists.

4.8 Integrative Negotiation

If the Environment Characterization (EC) approach in Section 4.7 cannot automatically construct a realizable specification φ_r', and the violations of $\varphi_{e,r}^t$ are due to a collaborative robot operating in the same workspace asynchronously, then the Integrative Negotiation approach (Alg. 4), first introduced in [88], may resolve the violation of the environment safety assumptions $\varphi_{e,r}'^t$ through modifying the environment (collaborative robot) behaviors.

The approach initiates an exchange of partial specifications between robot r and the robot in conflict with r, r_{env}, in the shared workspace. This algorithm assumes the violations of $\varphi_{e,r}^t$ are due to one environment robot. The specification φ_r^* in Alg. 4 is the same as the original specification φ_r when the controller execution begins, e.g., $\varphi_{s,r}^{*\,t} = \varphi_{s,r}^t$ and $\varphi_{e,r}^{*\,g} = \varphi_{e,r}^g$.

Negotiation is triggered by the robot detecting the conflict, assumed here to be robot r. Robot r notifies the conflicting robot r_{env} of the need to negotiate. The two robots in negotiation both execute Alg. 4, where in r_{env}'s perspective, it is r and robot r becomes r_{env}. Note that negotiation is only carried out once between any two robots.

During negotiation, the algorithm does not include any revisions to the environment safety assumptions from the EC approach – the addition of simple LTL conjunctions based on Eq. 4.3 and Eq. 4.4. Since the Integrative Negotiation approach changes the actual environment behaviors, the environment safety assumptions from the previous characterization $\varphi_{e,r}'^t$ may include environment behaviors that will no longer occur. The algorithm restores

Algorithm 4 Integrative Negotiation for participating robots

procedure IntegrativeNegotiation
2: Restore $\varphi_{e,r}^{\prime\, t}$ to $\varphi_{e,r}^{t}$
Send $\varphi_{e,r}^{t(r_{env})}$ and $\varphi_{s,r}^{g}$ to robot r_{env}
4: Receive $\varphi_{e,r_{env}}^{t(r)}$ and $\varphi_{s,r_{env}}^{g}$ from robot r_{env}
$\varphi_{s,r}^{t\,nego} \leftarrow \varphi_{s,r}^{*\,t} \wedge \varphi_{e,r_{env}}^{t(r)}$, $\varphi_{e,r}^{g\,nego} \leftarrow \varphi_{e,r}^{*\,g} \wedge \varphi_{s,r_{env}}^{g}$
6: $\varphi_{e,r}^{*\,i} = \bigwedge_{\pi \in x_{i+1,r}} \pi \wedge \bigwedge_{\pi \in \mathcal{X}_r \setminus x_{i+1,r}} \neg\pi$
$\varphi_{s,r}^{*\,i} = \bigwedge_{\pi \in \gamma_r^s(s_{i,r})} \pi \wedge \bigwedge_{\pi \in \mathcal{Y}_r \setminus \gamma_r^s(s_{i,r})} \neg\pi$
8: $\varphi_r^{nego} = \varphi_{e,r}^{*\,i} \wedge \varphi_{e,r}^{t} \wedge \varphi_{e,r}^{g\,nego} \rightarrow \varphi_{s,r}^{*\,i} \wedge \varphi_{s,r}^{t\,nego} \wedge \varphi_{s,r}^{g}$
$\mathcal{A}_r^{nego} \leftarrow$ SYNTHESIS(φ_r^{nego})
10: Send $\mathcal{A}_r^{nego}$ status to robot r_{env}. Wait for $\mathcal{A}_{r_{env}}^{nego}$ status
if φ_r^{nego} is realizable ($\mathcal{A}_r^{nego}$ is synthesized) **then**
12: $\mathcal{A}_r^{*} \leftarrow \mathcal{A}_r^{nego}$, $\varphi_r^{*} \leftarrow \varphi_r^{nego}$
else if $\varphi_{r_{env}}^{nego}$ is realizable (robot r_{env} can take robot r's task into consideration) **then**
14: $\varphi_r^{*} = \varphi_{e,r}^{*\,i} \wedge \varphi_{e,r}^{t} \wedge \varphi_{e,r}^{*\,g} \rightarrow \varphi_{s,r}^{*\,i} \wedge \varphi_{s,r}^{*\,t} \wedge \varphi_{s,r}^{g}$
$\mathcal{A}_r^{*} \leftarrow$ SYNTHESIS(φ_r^{*})
16: **else** (Both φ_r^{nego} and $\varphi_{r_{env}}^{nego}$ are unrealizable)
Abort execution. Analyze φ_r^{nego}. **return**
18: Send $\mathcal{A}_r^{*}$ status to robot r_{env}. Wait for $\mathcal{A}_{r_{env}}^{*}$ status
if φ_r^{*} or $\varphi_{r_{env}}^{*}$ is unrealizable **then**
20: Abort execution. Analyze the specifications. **return**
return $\mathcal{A}_r^{*}, \varphi_r^{*}$

the previous assumptions $\varphi_{e,r}^{\prime\, t}$ to the original environment safety assumptions $\varphi_{e,r}^{\prime\, t} = \varphi_{e,r}^{t}$ (line 2 in Alg. 4). If needed, environment characterization is performed after the integrative negotiation phase. We define safeties $\varphi_{e,r}^{t(r_{env})}$ as follows:

$$\varphi_{e,r}^{t(r_{env})} = \bigwedge \{\Box\, \psi_{e,r}^{t} \mid \psi_{e,r}^{t} \text{ is a clause in } \varphi_{e,r}^{t}, \exists\pi \text{ appears in } \psi_{e,r}^{t} \text{ we have that } \pi \in eAP_r \text{ and } \pi \text{ abstracts information about } r_{env}\}.$$

Each robot sends its environment assumptions relating to r_{env}, $\varphi_{e,r}^{t(r_{env})}$, and its system goals $\varphi_{s,r}^{g}$ to the environment robot r_{env} (line 3 in Alg. 4); each robot also receives the

environment assumptions relating to itself $\varphi^{t(r)}_{e,r_{env}}$ and the system goals $\varphi^{g}_{s,r_{env}}$ from the other robot (line 4 in Alg. 4).

Both robots attempt to incorporate assumptions from the other robot $\varphi^{t(r)}_{e,r_{env}}$ into their system safety guarantees $\varphi^{*\,t}_{s,r}$. Formally, the acquired assumptions $\varphi^{t(r)}_{e,r_{env}}$ are conjoined with the system safety guarantees $\varphi^{*\,t}_{s,r}$ and together they form the temporary new system guarantees $\varphi^{t\,nego}_{s,r}$ (line 5 in Alg. 4). If the resulting specification $\varphi^{nego}_{r_{env}}$ of the environment robot r_{env} is realizable, then robot r_{env} will satisfy the assumptions $\varphi^{t(r_{env})}_{e,r}$ robot r is making, and similarly for robot r.

Since robot r must maintain the safety guarantees $\varphi^{*\,t}_{s,r}$ when the environment specification $\varphi^{*}_{e,r}$ is satisfied, adding system safety guarantees $\varphi^{*\,t}_{s,r}$ further restrict the robot's behavior and may make the new specification unrealizable. To alleviate this situation, the system goals of the environment robot $\varphi^{g}_{s,r_{env}}$ are integrated into the new specification φ^{nego}_{r} as part of the environment liveness assumptions $\varphi^{g\,nego}_{e,r}$ (line 5 in Alg. 4). By doing so, we also restrict the set of environment behaviors the robot must respond to.

The initial conditions of the environment and the robot are updated, similar to Alg. 2, on lines 6 and 7 in Alg. 4. With the new specification φ^{nego}_{r} (line 8 in Alg. 4), the synthesis algorithm attempts to synthesize a new controller $\mathcal{A}^{nego}_{r}$ (line 9 in Alg. 4). Robot r then conveys the synthesis result of $\mathcal{A}^{nego}_{r}$ to the other robot r_{env} and waits for the synthesis status of $\mathcal{A}^{nego}_{r_{env}}$ from robot r_{env} (line 10 in Alg. 4).

4.8.1 Both robots can incorporate the other's specification

If the two new specifications φ_r^{nego} and $\varphi_{r_{env}}^{nego}$ are both realizable, then robot r expects no future violations of the environment safety assumptions $\varphi_{e,r}^{t}$ from robot r_{env}; likewise, robot r_{env} also anticipates no violations from robot r onwards. In this work, we guarantee that the robots will not experience any additional environment safety violations from the other robot. However, due to the decentralized approach, the robots may encounter livelock situations. Here, if the environment livenesses $\varphi_{e,r}^{g\,nego}$ of robot r are not satisfied, then from Eq. 4.1 robot r may or may not complete its system goals $\varphi_{s,r}^{g}$. If robot r does not satisfy its system goals $\varphi_{s,r}^{g}$, then robot r_{env} may not satisfy its system goals $\varphi_{s,r_{env}}^{g}$ either, since its specification $\varphi_{r_{env}}^{nego}$ now incorporates robot r's system goals into its environment livenesses $\varphi_{e,r_{env}}^{g\,nego}$ and robot r_{env} relies on robot r satisfying its goals. Practically speaking, since the robots are asynchronous, one robot could make more progress than the other and eventually resolves a livelock in many cases; however, in future work we will look into conditions for guaranteeing livelock free executions.

4.8.2 Only one robot can incorporate the other's specification

Even when only one of the two specifications is realizable, the robots may still continue their tasks. Consider robot r_{env} successfully synthesizes a new controller $\mathcal{A}_{r_{env}}^{nego}$ from the specification $\varphi_{r_{env}}^{nego}$ but robot r cannot from φ_r^{nego}. For robot r_{env}, this is identical to the scenario where both specifications are realizable: the new controller $\mathcal{A}_{r_{env}}^{nego}$ and specification $\varphi_{r_{env}}^{nego}$ replace the old controller $\mathcal{A}_{r_{env}}^{*}$ and specification $\varphi_{r_{env}}^{*}$. For robot r, receiving robot r_{env}'s successful status (line 13 in Alg. 4), robot r resynthesizes a new controller $\mathcal{A}_r^*$ from its specification φ_r^* before negotiation, with the initial conditions $\varphi_{e,r}^{*\,i}$ and $\varphi_{s,r}^{*\,i}$ replaced

(lines 14,15 in Alg. 4). In this case, robot r_{env} accommodates the assumptions of robot r which continues the mission with the previous specification φ_r^*.

Robot r then sends the latest synthesis result of $\mathcal{A}_r^*$ to robot r_{env} and waits for an update on the $\mathcal{A}_{r_{env}}^*$ status (line 18 in Alg. 4). If robot r synthesizes a controller $\mathcal{A}_r^*$ from the specification φ_r^*, then both robots resume their tasks with the latest automatons $\mathcal{A}_r^*$ and $\mathcal{A}_{r_{env}}^*$. The new controller $\mathcal{A}_r^*$ and specification φ_r^* are returned (line 21 in Alg. 4). Robot r continues its execution as before and satisfies its goals. Taking robot r's specification into account and knowing robot r can reach its goals, robot r_{env} can also satisfy its goals from Eq. 4.1. If φ_r^* is unrealizable, then the robots exit their execution and the specifications are analyzed (line 20 in Alg. 4). The synthesis results are exchanged twice (line 10, 18 in Alg. 4) because when $\varphi_{r_{env}}^{nego}$ is realizable and φ_r^{nego} is unrealizable, robot r will resynthesize a new controller with its original specification but with the current initial conditions (line 14 in Alg. 4). However, the specification could be unrealizable and both robots abort their tasks in this case. If φ_r^{nego} and $\varphi_{r_{env}}^{nego}$ are both realizable, then the second exchange is redundant. During the rest of the execution, robot r might violate robot r_{env}'s assumptions again, since robot r continues with its original specification.

4.8.3 Both robots cannot incorporate the other's specification

If both robots r and r_{env} cannot accommodate the other robot's mission when conducting their own task, i.e., φ_r^{nego} and $\varphi_{r_{env}}^{nego}$ are both unrealizable (line 16 in Alg. 4), then the algorithm provides specification analysis to the user, using the technique from [74] (line 17

in Alg. 4).

4.9 Computational implications

The synthesis process in this work consists of two steps: 1. computing a game structure $\mathcal{G}_r$ from a specification φ_r; and 2. extracting a strategy or controller $\mathcal{A}_r$ from the game.

In the Recovery approach (Section 4.6), we only modify the strategy extraction portion of the synthesis process. The computation of the structure $\mathcal{G}_r$ from a specification φ_r does not change, and computing a controller can still be done in time polynomial in the number of states as in [10].

In the Environment Characterization approach (Section 4.7), the total number of possible simple LTL conjunctions are $2^{|\mathcal{X}|} + 2^{2|\mathcal{X}|}$, with $2^{|\mathcal{X}|}$ due to Eq. 4.3 and $2^{2|\mathcal{X}|}$ due to the combination of any current inputs with any incoming inputs in Eq. 4.4. Since the approach only resynthesizes when a new environment is observed, this also translates into the maximum number of resynthesis steps possible.

In the Integrative Negotiation approach (Section 4.8), given a robot r can communicate with m other robots, the maximum number of resynthesis steps would be $2m$ for robot r. During negotiation, robot r first synthesizes a new controller by incorporating the other robot's specification. If it cannot do so, it resynthesizes a controller using its original specification with an updated initial condition. Robot r synthesizes at most twice in each negotiation.

Note that the complexity of all synthesis steps in the process together is the same as for standard GR(1) synthesis – exponential in the number of propositions and polynomial in the number of states in the game graph. This is because the number of resynthesis steps is bounded exponentially by the number of propositions (plus the number of other robots) by the argument above. Exponentially often executing an exponential synthesis step yields an exponential complexity.

Lemma 1. *With our work,*

(a) An infinite execution of a synthesized controller never violates the system safety guarantees $\varphi^t_{s,r}$.

(b) With the Environment Characterization (EC) approach, the environment safety assumptions $\varphi'^t_{e,r}$ converge based on the environment observations.

(c) The robot safety guarantees $\varphi^{*t}_{s,r}$ converge due to the Integrative Negotiation approach.

Proof. In this work, we leverage the synthesis technique from both [10] and [23].

Claim (a). This follows from the properties of a synthesized controller from [23]. A synthesized controller only contains transitions that satisfy both the environment specification $\varphi_{e,r}$ and the system specification $\varphi_{s,r}$. The Recovery approach still ensures that the synthesized controller satisfies the system guarantees $\varphi^t_{s,r}$, as the approach only adds new transitions (x, y, x', y') that are in ρ_s. The other two approaches resynthesize a new controller when the environment assumptions $\varphi^t_{e,r}$ are violated at runtime. The approaches either uphold the system guarantees $\varphi^t_{s,r}$ with the new controller or they abort the execution. The controller does not violate the system guarantees $\varphi^t_{s,r}$ during execution.

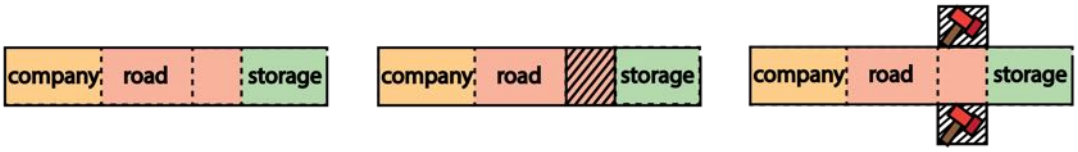
Claim (b). Consider the case where we observe a violation of the environment assumptions at runtime. With the EC approach, we update the original environment assumptions $\varphi^t_{e,r}$ with the newly-observed environment behaviors ($\varphi'^t_{e,r}$). Compared to $\varphi^t_{e,r}$, $\varphi'^t_{e,r}$ is now a more accurate description of the environment behaviors. As the execution continues, the robot can observe more environment behaviors. If every single environment behavior is possible, the environment assumptions $\varphi'^t_{e,r}$ would eventually converge to $\square$(**true**) by having a disjunct for every possible environment transition in the environment assumptions.

Claim (c). Consider the case where robot r detects a violation of its environment safety assumptions by a collaborative robot at runtime. With the Integrative Negotiation approach, robot r updates its system safety guarantees $\varphi^{*t}_{s,r}$ conjoining the environment robot r''s environment safety assumptions about r. This imposes additional system guarantees on robot r. After robot r exchanges its specification with all the collaborative robots around it, the robot specification $\varphi^*_{s,r}$ converges. $\square$

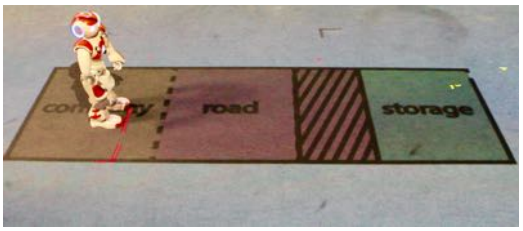
Thus, we are converging to a specification that better approximates the environment and describes the system, such that robots operating in a shared workspace can continue their tasks with guarantees.

4.10 Examples

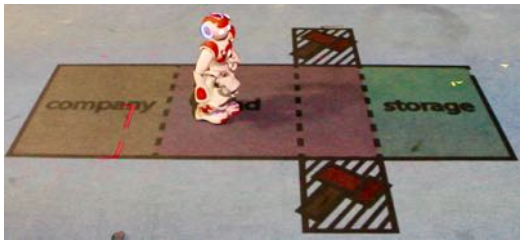
Figure 4.2: Workspace for Examples 2 and 4



(a) The workspace when: the door is open (left figure); the door is closed (middle), and the door is broken (right).

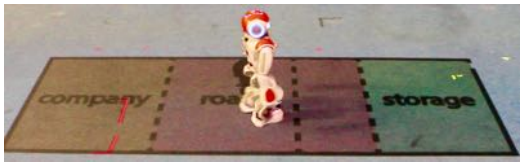


(b) Starting position of Alice.



(c) The door is broken when Alice is in *road*.

(d) The door is closed when Alice is in *road*.



(e) Once the door opens, Alice continues her task.

Figure 4.3: Alice, the orange Aldebaran Nao, performing her task as described in Example 11.

4.10.1 No Other Robot in the Workspace

Consider a patrol task for robot Alice (Example 11, Spec. 6) In this example, we only consider the three bottom left regions: *company*, *road* and *storage*. Executions of the examples can be found online[1].

Example 11. *Robot Alice conducts a patrol task and starts in company (line 1 in Spec. 6). When she receives an order, she visits storage and stays there (lines 4,7 in Spec. 6). If she does not receive an order or the order ends, she goes to company and stays there (lines 5,6*

[1] `https://youtu.be/Uw_bN2hVe1I`

in Spec. 6). Between the regions road and storage, there is a door. Alice can visit storage if the door is open or broken (line 3 in Spec. 6). The environment assumption is that the door is open when Alice is in road or storage (line 2 in Spec. 6). Fig. 4.3a shows when the door is open, closed or broken.

Specification φ_A is realizable and a controller $\mathcal{A}_A$ is successfully synthesized.

Specification 6 Alice's task φ_A in Example 11

$$
\begin{aligned}
1\ \varphi^{i}_{s,A} &= \pi_{company,A} \qquad \varphi^{i}_{e,A} = \pi^{c}_{company,A} \\
2\ \varphi^{t}_{e,A} &= \square((\bigcirc \pi^{c}_{road,A} \vee \bigcirc \pi^{c}_{storage,A}) \rightarrow (\bigcirc \pi_{doorOpen}) \\
3\ \varphi^{t}_{s,A} &= \square(\neg(\bigcirc\pi_{doorOpen} \vee \bigcirc\pi_{doorBroken}) \rightarrow \neg \bigcirc \pi_{storage,A}) \wedge \\
4 &\qquad \square((\pi_{order,A} \wedge \pi^{c}_{storage,A}) \rightarrow \bigcirc\pi_{storage,A}) \wedge \\
5 &\qquad \square((\neg\pi_{order,A} \wedge \pi^{c}_{company,A}) \rightarrow \bigcirc\pi_{company,A}) \wedge \\
6\ \varphi^{g}_{s,A} &= \square \diamond(\neg\pi_{order,A} \rightarrow (\pi_{company,A} \wedge \pi^{c}_{company,A})) \wedge \\
7 &\qquad \square \diamond(\pi_{order,A} \rightarrow (\pi_{storage,A} \wedge \pi^{c}_{storage,A}))
\end{aligned}
$$

4.10.1.1 Environment Characterization only (1^{st} example in the online video[1])

The task starts (Fig. 4.3b) and Alice receives an order. Alice heads to *storage* and the door is now broken. In the *road* region, Alice detects a violation of her assumption that 'the door should be open when she is in *road* or *storage*' (line 2 in Spec. 6, Fig. 4.3c). This triggers Environment Characterization and the algorithm appends the latest observed environment behaviors into Alice's updated specification, φ'_A, and resynthesizes a new controller $\mathcal{A}'_A$. In this case, the observed environment behavior is $(\neg \bigcirc \pi_{doorOpen} \wedge \bigcirc \pi_{doorBroken} \wedge \bigcirc \pi_{order,A} \wedge \bigcirc \pi^{c}_{road,A})$, with only the incoming valuation of the environment propositions as shown in Eq. 4.3.

Since Alice can still proceed to *storage* with the door broken thus reaching her goal, the revised specification is realizable and a new controller is synthesized. Alice continues her way to *storage*. When Alice is in *storage*, the previously added simple LTL conjunction is invalid as Alice is no longer in *road*. The algorithm adds another conjunction ($\neg \bigcirc \pi_{doorOpen} \wedge \bigcirc \pi_{doorBroken} \wedge \bigcirc \pi_{order,A} \wedge \bigcirc \pi^{c}_{storage,A}$) to Alice's environment safety assumptions $\varphi'^{\,t}_{e,A}$, and it generates a new controller for Alice.

Once the order ends, Alice heads back to *company*. The algorithm detects a new environment behavior and appends its observation into Alice's specification: ($\neg \bigcirc \pi_{doorOpen} \wedge \bigcirc \pi_{doorBroken} \wedge \neg \bigcirc \pi_{order,A} \wedge \bigcirc \pi^{c}_{storage,A}$). However, the specification φ'_A is not realizable as Alice cannot move to *company* eventually with the current conjunctions and the activation-completion paradigm.

Since adding a simple LTL conjunction created with Eq. 4.3 cannot render the specification realizable, the algorithm appends instead a conjunction created with Eq. 4.4: $((\neg\pi_{doorOpen} \wedge \pi_{doorBroken} \wedge \pi_{order,A} \wedge \pi^{c}_{storage,A})) \wedge (\neg \bigcirc \pi_{doorOpen} \wedge \bigcirc \pi_{doorBroken} \wedge \neg \bigcirc \pi_{order,A} \wedge \bigcirc \pi^{c}_{storage,A}))$ to the specification. This conjunction includes information about both the current and incoming environment propositions. We switch to $\varphi^{t,EC2}_{e,A}$ from $\varphi^{t,EC1}_{e,A}$ and the environment assumptions now encode more precise observation of the environment. With the new environment assumptions, it is explicitly encoded that the door will stay broken and Alice can continue on her way to *company*. The new specification is now realizable, a new controller is synthesized and Alice's execution resumes. Alice continues and resynthesizes when she observes new environment behaviors and eventually goes back to *company*.

Alice then receives an order again but this time the door is closed. Again, the algorithm

detects a violation of the assumption that 'the door is open when Alice is in *road* or *storage*' (Fig. 4.3d). The algorithm appends a new conjunction to the specification but the specification is unrealizable, since if the door is closed forever, then Alice can never reach *storage*. The algorithm provides feedback to the user, highlighting the portion of the specification that leads to unrealizability and asking for a user input. At this point, the user can either abort the task and revise the entire specification, or if they know more about how the environment behaves, they can add environment livenesses to the specification.

In this case, the user adds an environment liveness $\square \diamondsuit(\pi_{doorOpen} \vee \pi_{doorBroken})$ that states the door will eventually be open or broken. The synthesis algorithm successfully creates a new controller. The execution resumes and Alice waits until the door is open (Fig. 4.3e) and then visits *storage*.

4.10.1.2 Recovery only (2^{nd} example in the online video[1])

Synthesizing Spec. 6 with the recovery approach provides robustness to temporary environment safety violations. The controller synthesized without Recovery transitions contains 73 states while the one synthesized with Recovery transitions contains 154 states.

With the 'recovery' controller, when Alice detects that the door is not open but broken (violation), she can continue her task without resynthesis. When the door is closed, however, the controller relies on the fact that the environment safety assumptions $\varphi^t_{e,A}$ will eventually hold. If that is not the case, Alice will wait in *road* forever without providing feedback. Combining both approaches, as described next, eliminates unnecessary resynthesis steps.

4.10.2 Example with Recovery and Environment Characterization (3^{rd} example in the online video[1])

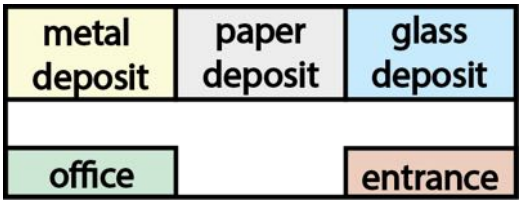
The two approaches can be used together by executing a 'recovery' controller from the start. The Environment Characterization algorithm appends all newly-observed environment behaviors to the specification during execution. The user specifies either the duration of the violation or the number of state transitions before controller resynthesis is triggered. This reduces resynthesis steps while maintaining robustness to assumption violations.

Example 12. *Consider the workspace shown in Fig. 4.4a where a robot patrols between office and entrance. When it senses an alarm, it goes to the office and notifies the manager. The manager then acknowledges that they received the warning by tapping the head of the robot. Also, if the robot senses a spill when it is patrolling, it stops and calls for help.*

In addition to patrolling, the robot also conducts a recycling task. If the robot is given metal, glass or paper, it will carry the item to the corresponding deposit region.

The user makes three assumptions in the specification: 1. the manager will turn off the alarm before acknowledging the robot; 2. alarm and spill will not happen at the same time; and 3. in a deposit region, the robot will not see an item that does not belong to the corresponding deposit region.

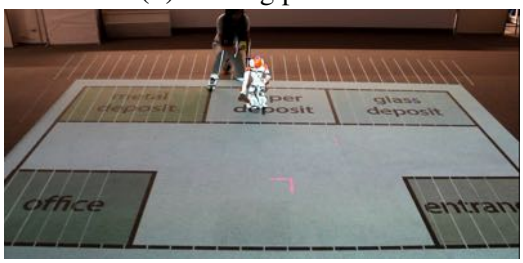
The user creates a realizable LTL specification for this example and synthesizes a 'recovery' controller. In this example, a controller is resynthesized if an environment safety assumption violation continues for more than 150 state transitions.



(a) Workspace of Example 12.

(b) Starting position.

(c) Robot picks up a paper item.



(d) Robot detects a metal item in the paper_deposit.

(e) Robot detects a spill.

(f) Manager dismisses Robot.

Figure 4.4: The robot performing the task described in Example 12.

During execution, when the robot is patrolling, it detects a paper item (Fig. 4.4c). The robot picks it up and brings it to the paper deposit region to drop it off. When dropping off the item, it observes a metal item that is not supposed to be there (Fig. 4.4d), violating the third assumption. Using the 'recovery' controller, the robot can continue the task without resynthesis and it heads to the metal deposit region and drops off the item.

As the robot continues patrolling, it detects an alarm and heads to the office to inform the manager. On the way to the office, it also observes a spill (Fig. 4.4e). This violates the second assumption that the robot only detects one of the two incidents at a time. With the

Recovery approach, the robot waits to see if the violation is temporary. Since the violation lasts longer than the user-defined threshold, the Environment Characterization rewrites and resynthesizes the specification but the updated specification is unrealizable.

In this case, our system provides feedback and prompts the user for input to resolve the violation. The user then decides to include an environment liveness that the spill will eventually go away. The new specification is realizable and the robot waits until the spill clears.

Once the spill is removed, the robot continues on its way to the office and reports to the manager. After the report, before turning off the alarm, the manager dismisses the robot by tapping its head (Fig. 4.4f). This violates the first assumption. In this case the violation is temporary, thus the 'recovery' controller can handle it without need for resynthesis.

4.10.3 Communicating Robots in the Workspace (4^{th} example in the online video [1])

Consider a three-robot scenario (Example 13): DeliveryAgent ('D'), KitchenAssistant ('A') and Chef ('C'), operating in the workspace in Fig 4.2. The propositions in eAP_r for $r \in \{D, A, C\}$ are in bold for clarity.

Example 13. *Robot D starts in company (line 1 in Spec. 7, and line 2 in Spec. 7, 8 and 9). When D receives an order for ingredients, it picks up the ingredients at company (line 7 in Spec. 7) and delivers the ingredients to storage (line 3, 8-12, 14 in Spec. 7). Since D cannot go to storage when A is not opening the door (line 13 in Spec. 7), D assumes that A*

will open the door when D is in road or storage (line 6 in Spec 7). It returns to company when the delivery is done (line 4, 15 in Spec. 7).

Robot A is new to the workforce; it has capabilities to pass ingredients or to open the door but it does not know when to perform its actions. It only knows that it does not pass ingredients when no ingredients are present (line 5 in Spec. 8). Robot C assumes A passes ingredients when they arrive (line 5 in Spec. 9) and C then cooks any ingredients received from A (line 6-12 in Spec. 9). The robots C and A always stay in place (line 5 in Spec. 7, line 3 in Spec. 8 and 9); robot D never enters the kitchen, which consists of the regions prepArea and cookingArea (line 4 in Spec. 8 and 9).

As an example, for D, $Reg_D = \{\pi_{company,D}, \pi_{road,D}, \pi_{storage,D}\}$ and similarly for Reg^c_D; $eAP_D = \{\boldsymbol{\pi_{openDoor,A}}, \boldsymbol{\pi^c_{cookingArea,C}}, \boldsymbol{\pi^c_{prepArea,A}}\}$; $Act_D = \{\pi_{deliver,D}, \pi_{pickup,D}\}$; $Sen_D = \{\pi_{order,D}\}$; $Mem_D = \{\pi_{orderReceived,D}\}$.

Specification 9 Chef's task φ_C for Example 13

1 $\varphi^i_{s,C} = \pi_{cookingArea,C}$

2 $\varphi^i_{e,C} = \pi^c_{cookingArea,C} \wedge \boldsymbol{\pi^c_{prepArea,A}} \wedge \boldsymbol{\pi^c_{company,D}}$

3 $\varphi^t_{e,C} = \Box(\bigcirc \boldsymbol{\pi^c_{prepArea,A}}) \wedge$

4 $\quad \Box \neg(\bigcirc \boldsymbol{\pi^c_{cookingArea,D}} \vee \bigcirc \boldsymbol{\pi^c_{prepArea,D}}) \wedge$

5 $\quad \Box(\bigcirc \pi_{ingredientArrived} \rightarrow \bigcirc \boldsymbol{\pi_{passIngredient,A}})$

$\varphi^g_{s,C} = \Box \Diamond((\pi_{ingredientArrived} \vee \pi_{receivedIngredient,C}) \rightarrow (\pi_{cooking,C} \wedge \pi^c_{cooking,C}))$

6 $\varphi^g_{e,C} = \Box \Diamond(\boldsymbol{\pi_{passIngredient,A}} \rightarrow \boldsymbol{\pi^c_{passIngredient,A}})$

$\varphi^t_{s,C} = \Box((\bigcirc \boldsymbol{\pi^c_{passIngredient,A}} \wedge \neg \pi^c_{cooking,C}) \rightarrow \bigcirc \pi_{receivedIngredient,C}) \wedge$

7 $\quad \Box(\pi^c_{cooking,C} \rightarrow \neg \bigcirc \pi_{receivedIngredient,C}) \wedge$

$\quad \Box((\pi_{ingredientArrived,C} \wedge \neg \pi^c_{cooking,C}) \rightarrow \bigcirc \pi_{receivedIngredient,C}) \wedge$

$\quad \Box((\neg \pi_{receivedIngredient,C} \wedge \neg \bigcirc \boldsymbol{\pi^c_{passIngredient,A}}) \rightarrow \neg \bigcirc \pi_{receivedIngredient,C}) \wedge$

8 $\quad \Box(\bigcirc \pi_{receivedIngredient,C} \leftrightarrow \bigcirc \pi_{cooking,C})$

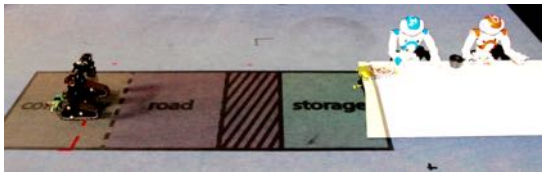
Specification 7 Delivery agent's task φ_D in Example 13

1 $\varphi^i_{s,D} = \pi_{company,D}$
2 $\varphi^i_{e,D} = \pi^c_{company,D} \wedge \boldsymbol{\pi^c_{cookingArea,C}} \wedge \boldsymbol{\pi^c_{prepArea,A}}$
3 $\varphi^g_{s,D} = \square \diamond(\neg\pi_{orderReceived,D} \rightarrow (\pi_{company,D} \wedge \pi^c_{company,D})) \wedge$
4 $\square \diamond(\pi_{orderReceived,D} \rightarrow (\pi_{storage,D} \wedge \pi^c_{storage,D}))$
5 $\varphi^t_{e,D} = \square(\bigcirc \boldsymbol{\pi^c_{cookingArea,C}}) \wedge \square(\bigcirc \boldsymbol{\pi^c_{prepArea,A}}) \wedge$
6 $\square((\bigcirc \pi^c_{road,D} \vee \bigcirc \pi^c_{storage,D}) \rightarrow (\bigcirc \boldsymbol{\pi_{openDoor,A}}))$
7 $\varphi^t_{s,D} = \square((\bigcirc \pi_{order,D} \wedge \neg \bigcirc \pi_{orderReceived,D}) \rightarrow \bigcirc \pi_{pickup,D}) \wedge$
$\square(((\bigcirc \pi_{order,D} \wedge \pi^c_{pickup,D}) \wedge \neg \pi^c_{deliver,D}) \rightarrow \bigcirc \pi_{orderReceived,D}) \wedge$
8 $\square((\pi^c_{deliver,D} \rightarrow \neg \bigcirc \pi_{orderReceived,D}) \wedge$
9 $\square((\pi_{orderReceived,D} \wedge \neg \pi^c_{deliver,D}) \rightarrow \bigcirc \pi_{orderReceived,D})$
$\square((\neg\pi_{orderReceived,D} \wedge \neg(\bigcirc \pi_{order,D} \wedge \pi^c_{pickup,D})) \rightarrow \neg \bigcirc \pi_{orderReceived,D}) \wedge$
10 $\square((\bigcirc \pi_{orderReceived,D} \wedge \bigcirc \pi^c_{storage,D}) \leftrightarrow \bigcirc \pi_{deliver,D}) \wedge$
11 $\square(\neg \bigcirc \boldsymbol{\pi_{openDoor,A}} \rightarrow \neg \bigcirc \pi_{storage,D}) \wedge$
12 $\square((\pi_{orderReceived,D} \wedge \pi^c_{storage,D}) \rightarrow \bigcirc \pi_{storage,D}) \wedge$
13 $\square((\neg\pi_{orderReceived,D} \wedge \pi^c_{company,D}) \rightarrow \bigcirc \pi_{company,D})$

Specification 8 Kitchen assistant's task φ_A in Example 13

1 $\varphi^i_{s,A} = \pi_{prepArea,A}$
2 $\varphi^i_{e,A} = \pi^c_{prepArea,A} \wedge \boldsymbol{\pi^c_{cookingArea,C}} \wedge \boldsymbol{\pi^c_{company,D}}$
3 $\varphi^t_{e,A} = \square(\bigcirc \boldsymbol{\pi^c_{cookingArea,C}}) \wedge$
4 $\square \neg(\bigcirc \boldsymbol{\pi^c_{cookingArea,D}} \vee \bigcirc \boldsymbol{\pi^c_{prepArea,D}})$
$\varphi^t_{s,A} = \square((\neg\pi_{ingredientArrived} \wedge \pi_{prepArea,A}) \rightarrow \neg \bigcirc \pi_{passIngredient,A})$

The three specifications are all realizable and the robots start executing their controllers (Fig. 4.5a). First, D receives an order, picks up the ingredients and proceeds to *storage*. On its way, it notices that the door is closed and this prevents its entry to *storage* (Fig. 4.5b). A violates D's assumption that 'when you are in *road* or *storage*, A opens the door' (line 6 in Spec. 7). D triggers Environment Characterization but appending the new environment behaviors results in an unrealizable specification. Knowing robot A can control the door, D



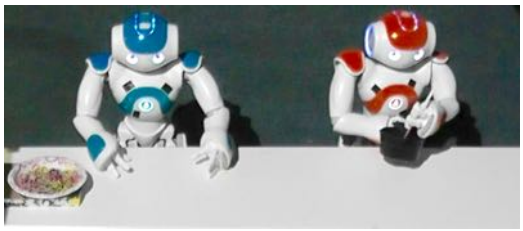
(a) Starting positions of the three robots.

(b) D heads to *storage* with ingredients.

(c) A opens the door.

(d) D delivers the ingredients.

(e) A passes ingredients to C.



(f) C cooks the ingredients.

Figure 4.5: DeliveryAgent ('D'), the Johnny5 robot; KitchenAssistant ('A'), the blue Aldebaran Nao, and Chef ('C'), the orange Aldebaran Nao, performing their tasks as described in Example 13.

initiates the Integrative Negotiation approach with A.

In this case, A can incorporate D's assumption that it should open the door when D is in *road* or *storage* (line 6 in Spec. 7). This is added as a system guarantee $\varphi^t_{s,A}$ to A's specification: $\varphi^t_{s,A} = \varphi^t_{s,A} \wedge \square((\bigcirc \boldsymbol{\pi^c_{road,D}} \vee \bigcirc \boldsymbol{\pi^c_{storage,D}}) \rightarrow (\bigcirc \pi_{openDoor,A})$. Both revised specifications are realizable and updated controllers are synthesized.

With the door opened, D enters *storage* and delivers the ingredients (Fig. 4.5d). Both C and A now know that the ingredients have arrived. However, to cook, C requires A to pass the ingredients (line 5 in Spec. 9). Since A is not doing so, C negotiates with A and asks it to pass the ingredients when they arrive. A modifies its system safety guarantees $\varphi^t_{s,A}$ to

$\varphi^t_{s,A} = \varphi^t_{s,A} \wedge \square(\bigcirc \pi_{receivedIngredient} \rightarrow \bigcirc \pi_{passIngredient,A})$.

The Integrative Negotiation is successful and A passes the ingredients to C (Fig. 4.5e). Once C receives the ingredients, it cooks the ingredients (Fig. 4.5f). D has finished its delivery task and it returns to *company*.

4.10.4 Comparison

We compare the relative resilience of the approaches presented in this work by examining the different examples.

Example 11

- **Base Case:** With only a controller synthesized from [23], the controller fails immediately when the door is broken.
- **Environment Characterization (EC):** Out of the four assumption violations detected in the example, we resynthesize a controller successfully twice with Eq. 4.3 and once with Eq. 4.4 before the update with the observed environment behaviors renders the specification unrealizable, and we employ the technique from [74] to provide feedback.
- **Recovery:** With this approach, the synthesized controller contains 154 states compared to 73 states when using [23]. By including these extra states and transitions, Alice can make 6 transitions in her controller with the door broken until the violation is resolved when Alice goes back to *company*.

Example 12

We can reduce the number of resyntheses leveraging both the Recovery approach and the EC approach. With only the EC approach, we would resynthesize a new controller three times; while leveraging both the Recovery approach and the EC approach, we only resynthesize a new controller once.

Example 13

- **Base Case:** With the synthesized controller using [23], when D is in *road* and the door is closed, it would abort its mission.
- **Recovery:** D would wait infinitely long in front of the door when the door is closed.
- **EC:** D updates its specification but it cannot synthesize a new controller. The user pauses the execution of both D and A, changes both specifications manually and resynthesizes two new controllers. This can only be done if the user can control both robots. Otherwise, D still aborts its mission.
- **Integrative Negotiation:** D can modify its environment through communication with A in the shared workspace. D can finish its delivery task.

The *Integrative Negotiation* increases the chance that a robot can maintain its guarantees during execution. However, the challenge increases with the number of agents in the workspace and the number of negotiations conducted. It is in general harder to successfully finish the Integrative Negotiation approach with more agents as it implicitly assigns priorities to the agents. Depending on the priority queue, it can result in different outcomes and the robots may or may not have controllers.

4.11 Evaluation

In this section, we qualitatively evaluate the three approaches proposed in this work.

Recovery: As discussed in Section 4.9, the synthesis complexity using the Recovery approach remains the same as the original algorithm in [10]. The approach adds in 'recovery' transitions which allows the robot to continue its task when an environment assumption is violated. However, the approach does rely on the violation being resolved eventually. A robot can be stuck forever if the violation persists. We advise using the approach alongside an assumption monitor that keeps track of violations at runtime and triggers termination when a violation persists.

Environment Characterization (EC): The EC approach updates the user specification automatically at runtime based on any newly observed environment behaviors and resynthesizes a new controller. Even though the EC approach preserves the most up-to-date observations about the environment, the resynthesis process can be time-consuming as the number of propositions increases. Also, if the environment behavior is random, we should not execute this approach at runtime as it permanently updates the specification with random observations in this case. We advise using this approach when the controller synthesis time, which depends on the number of propositions and the specification, is less than 1 minute. Resynthesis is relatively frequent in this approach so any synthesis time longer than a minute can make the approach infeasible. An alternative for specifications with large amount of propositions is to execute the EC approach with the Recovery approach such that we update the specification with new environment behaviors at runtime, but we only

resynthesize a new controller when the robot is not making progress towards its goals.

Integrative Negotiation: The Integrative Negotiation approach initiates communication between a pair of conflicting robots. The robots then modify their behaviors such that both robots can finish their tasks. It is challenging to use the approach when a conflict involves more than two robots. In this case, preprocessing is needed to separate the group of robots into pairs before carrying out the approach. The outcome of the negotiation can also be different depending on the order of negotiations between multiple pairs of robots. The chance of successful negotiation in general decreases as the number of robots operating in the shared workspace increases. The approach is analogous to creating a priority queue at runtime. One robot may defer its actions so that another robot can complete its task before the robot continues. It is more challenging to insert another robot into the queue as the number of robots increases. With more robots to consider, the robot may not be able to finish its task taking into account the task of the other robots and the negotiation would fail. We advise using the approach when it is a two-robot conflict and when the communication between the robots is stable, so that the robots can receive the full information exchange.

In the future, quantitative analysis and evaluation of the three approaches would be useful for comparison with other similar approaches.

4.12 Summary

In this work, we have explored different approaches to formally tackle unexpected environment behaviors that are detected during the execution of provably-correct controllers,

thereby creating controllers that are robust to violations of assumptions about the environment. We argue that every robot controller is created with underlying assumptions about the environment, whether implicit or explicit, and our contribution is in developing formal techniques that can automatically adjust the controller in response to violations of those assumptions while maintaining task guarantees, whenever possible.

The Recovery approach in Section 4.6 tackles the problem offline through adding extra transitions and states in the controller before the execution begins; the Environment Characterization in Section 4.7 resolves the issue online through appending newly observed environment behaviors to the specification; the Integrative Negotiation initiates an exchange of specification between two robots if the conflict is caused by another robot in the same workspace. These approaches create fallback for abnormal events during execution. Previously without our approaches, the controller synthesized with [10] results in unknown execution when the environment assumptions are violated at runtime. The robot may not complete its task but here our approaches allow the robot to continue its task safely when violations occur. Towards the end of the work, we have also showed how the three approaches can be combined and used together in the Examples section. We furthermore compare our controllers with an original controller from [10] in the Section.

The Integrative Negotiation approach is currently limited to two robots at a time and cannot deal with a multi-way conflict, i.e, a conflict involving three robots or more. In future work we will explore whether it is possible to decompose a multi-robot conflict into pairwise conflicts and investigate approaches when such decomposition is not possible.

CHAPTER 5

CONCLUSION

Robotics technology is undergoing rapid development. The improvement in both robotic software and hardware has enabled robots to conduct tasks outside of organized factories and operate in unpredictable and challenging environments. With these changes in operating environments, robots face new challenges in coping with unexpected scenarios before and during execution. It is crucial that robot execution is robust against anomalies. This thesis has investigated different anomalies that can arise in robot task execution and presented solutions to improve task execution.

Chapter 2 presents an approach to transfer programs between robots when a source robot is no longer available. The approach leverages the standardized framework of the Robot Operating System (ROS). It divides the problem into finding the correct communication channels and modifying robot commands to create similar behaviors. These behaviors include mobile-base movement, joint trajectory planning and path planning. The approach allows another robot with similar capabilities to continue a time-critical task with reduced delay. The approach can furthermore reduce the development time of robotic software when researchers and engineers can reuse robot programs easily. Besides the automatic aspect, the approach keeps the user in the loop by providing feedback on the transfer result which allows verification by the user before execution. Future work on robot program transfer includes finding correlations among different communication channels for better channel replacements. The transfer of robot programs from one robot with one kinematic model to another robot with another model, such as from an UAV to a mobile base robot,

also poses challenge as it requires finding similarities in the robot pairs. Another direction is leveraging symbolic or concolic testing techniques to obtain commands generated from sensors for command analysis and replacement.

Besides transferring and executing these robot programs, which are also called the low-level robot controllers, these controllers usually execute together with high-level controllers to form hybrid controllers. The high-level controllers are automatically generated from user commands and the low-level controllers communicate and send commands with the actual robot. Users often deploy these hybrid controllers on robots, but the interactions between low-level and high-level controllers are rarely analyzed and can result in erratic execution.

Chapter 3 starts by describing a framework that naturally connects controllers synthesized from high-level task specifications with low-level robot programs using ROS. With the framework, users can then analyze the interaction of low-level controllers which in return lead to modifications of the high-level specifications. The approach ensures that the high-level task specifications take into account the behaviors of the low-level controllers automatically. In the past, the low-level behaviors were only encoded in the high-level task specification based on user insight. Future work includes analyzing interactions among robots each with a controller synthesized from a high-level task specification. Another direction for future study is suggesting high-level specification changes based on interactions of robot sensors.

Finally, Chapter 4 illustrates approaches in response to unexpected environment events in terms of environment assumption violations, during the execution of high-level provably-correct controllers. The chapter outlines three approaches to maintain robot safety and

allow the robot to continue and finish its task. The Recovery approach adds in recovery transitions during the synthesis of robot controllers, such that the robot can still proceed to its goals during violations as long as the robot is safe. The Environment Characterization approach automatically modifies the specification at runtime to incorporate any newly observed environment behaviors into the specification. The unexpected environment can also involve other robots working in the shared workspace. In this case, the Integrative Negotiation approach allows any two robots in conflict to exchange and modify their specifications. This changes the environment behaviors created by the other robot and allows both robots to complete their tasks. Previously without our approaches, the controller execution is unknown after a violation in [10]. The robot may not finish its task and the controller cannot guarantee robot safety. Here our work allows the robot to continue its task with safety guarantees when violations occur. If the robot cannot continue its task safely, we abort execution and provide feedback to the user. Future work includes prioritizing and sequencing robot negotiations when conflicts involve more than two robots. Another possible future study is exploring the removal or addition of robot goals as another robot enters or leaves a shared workspace.

BIBLIOGRAPHY

[1] Alexandre Albore and Piergiorgio Bertoli. Safe LTL Assumption-Based Planning. In *ICAPS*, pages 193–202, 2006.

[2] Javier Alonso-Mora, Stuart Baker, and Daniela Rus. Multi-robot formation control and object transport in dynamic environments via constrained optimization. *I. J. Robotics Res.*, 36(9):1000–1021, 2017.

[3] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *FMCAD*, 2013.

[4] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional synthesis of reactive controllers for multi-agent systems. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 251–269, 2016.

[5] Tim Baarslag and Koen V. Hindriks. Accepting optimally in automated negotiation with incomplete information. In *AAMAS*, 2013.

[6] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins, and George J. Pappas. Symbolic planning and control of robot motion [Grand Challenges of Robotics]. *IEEE Robot. Automat. Mag.*, 14(1), 2007.

[7] Amit Bhatia, Lydia E. Kavraki, and Moshe Y. Vardi. Sampling-based motion planning with temporal goals. In *ICRA*, pages 2689–2696, 2010.

[8] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.

[9] Roderick Bloem, Rüdiger Ehlers, and Robert Könighofer. Cooperative reactive synthesis. In *Automated Technology for Verification and Analysis - International Symposium, ATVA*, pages 394–410, 2015.

[10] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3), 2012.

[11] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield Synthesis: - Runtime Enforcement for Reactive Systems. In *TACAS*, 2015.

[12] J. Bohren and S. Cousins. The smach high-level executive [ros news]. *IEEE Robotics Automation Magazine*, 17(4):18–20, Dec 2010.

[13] ROSwiki. Vicon Bridge. `http://wiki.ros.org/vicon_bridge` [2016].

[14] José Cano, Alejandro Bordallo, Vijay Nagarajan, Subramanian Ramamoorthy, and Sethu Vijayakumar. Automatic configuration of ROS applications for near-optimal performance. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, 2016.

[15] Yushan Chen, Jana Tumova, and Calin Belta. LTL robot motion control based on automata learning of environmental dynamics. In *ICRA*, 2012.

[16] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artif. Intell.*, 147(1-2):35–84, July 2003.

[17] Alessandro Cimatti and Marco Roveri. Conformant Planning via Symbolic Model Checking. *CoRR*, abs/1106.0252, 2011.

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

[19] William Cushing and Subbarao Kambhampati. Replanning: a new perspective. In *Poster of ICAPS*, 2005.

[20] Neil T. Dantam, Zachary K. Kingston, Swarat Chaudhuri, and Lydia E. Kavraki. Incremental Task and Motion Planning: A Constraint-Based Approach. In *RSS*, 2016.

[21] Pedro Henrique de Rodrigues Quemel e Assis Santana and Brian Charles Williams. Chance-Constrained Consistency for Probabilistic Temporal Plan Networks. In *ICAPS*, 2014.

[22] Rüdiger Ehlers. Generalized Rabin(1) Synthesis with Applications to Robust System Synthesis. In *NASA Formal Methods NFM. Proceedings*, 2011.

[23] Rüdiger Ehlers, Robert Könighofer, and Roderick Bloem. Synthesizing cooperative reactive mission plans. In *IROS*, 2015.

[24] Rüdiger Ehlers and Vasumathi Raman. Slugs: Extensible gr(1) synthesis. In *CAV*, 2016.

[25] Rüdiger Ehlers and Ufuk Topcu. Resilience to Intermittent Assumption Violations in Reactive Synthesis. In *HSCC*, pages 203–212, 2014.

[26] Esra Erdem, Kadir Haspalamutgil, Can Palaz, Volkan Patoglu, and Tansel Uras. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In *ICRA*, 2011.

[27] S. Shaheen Fatima, Michael Wooldridge, and Nicholas R. Jennings. An agenda-based framework for multi-issue negotiation. *Artif. Intell.*, 152, 2004.

[28] Jie Fu, Herbert G. Tanner, and Jeffrey Heinz. Adaptive planning in unknown environments using grammatical inference. In *CDC*, 2013.

[29] Fausto Giunchiglia and Paolo Traverso. Planning As Model Checking. In *Proceedings of the European Conference on Planning: Recent Advances in AI Planning*, ECP '99, pages 1–20, 2000.

[30] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*, chapter Introduction. John Wiley & Sons, Ltd, 2000.

[31] Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11. ACM, 2011.

[32] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 14th International Symposium on*. IEEE, 2012.

[33] Jason Hardy and Mark E. Campbell. Contingency Planning Over Probabilistic Obstacle Predictions for Autonomous Road Vehicles. *IEEE Trans. Robotics*, 29(4):913–929, 2013.

[34] Keliang He, Morteza Lahijanian, Lydia E. Kavraki, and Moshe Y. Vardi. Towards manipulation planning with temporal logic specifications. In *ICRA*, pages 346–352, 2015.

[35] Frederik W. Heger and Sanjiv Singh. Robust robotic assembly through contingencies, plan repair and re-planning. In *ICRA*, pages 3825–3830, 2010.

[36] Qiang Huang, Kazuhito Yokoi, Shuuji Kajita, Kenji Kaneko, Hirohiko Arai, Noriho Koyachi, and Kazuo Tanie. Planning walking patterns for a biped robot. *IEEE Trans. Robotics and Automation*, 17:280–289, 2001.

[37] Austin Jones, Zhaodan Kong, and Calin Belta. Anomaly detection in cyber-physical systems: A formal methods approach. In *IEEE CDC*, 2014.

[38] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014.

[39] Sertac Karaman and Emilio Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *CDC*, 2009.

[40] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):664–676, April 2017.

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

[42] Antonín Komenda, Peter Novák, and Michal Pechoucek. Domain-independent multi-agent plan repair. *J. Network and Computer Applications*, 37, 2014.

[43] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD*, pages 152–159, 2009.

[44] H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. Temporal Logic based Reactive Mission and Motion Planning. *IEEE T-RO*, 2009.

[45] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Where's Waldo? Sensor-Based Temporal Logic Motion Planning. In *ICRA*, 2007.

[46] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[47] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE T-RO*, 25, 2009.

[48] Orna Kupferman and Moshe Y. Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[49] Morteza Lahijanian, Matthew R. Maly, Dror Fried, Lydia E. Kavraki, Hadas Kress-Gazit, and Moshe Y. Vardi. Iterative Temporal Planning in Uncertain Environments With Partial Satisfaction Guarantees. *IEEE T-RO*, 2016.

[50] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Learning repetitive text-editing procedures with smartedit. *Your Wish Is My Command: Giving Users the Power to Instruct Their Software*, 2001.

[51] M. Lauer, M. Amy, J. C. Fabre, M. Roy, W. Excoffon, and M. Stoicescu. Engineering adaptive fault-tolerance mechanisms for resilient computing on ros. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 94–101, Jan 2016.

[52] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[53] Vu Le and Sumit Gulwani. FlashExtract: a framework for data extraction by examples. In Michael F. P. O'Boyle and Keshav Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*. ACM, 2014.

[54] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In

Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on. IEEE, 2016.

[55] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[56] A Solar Lezama. *Program synthesis by sketching*. PhD thesis, PhD thesis, EECS Department, University of California, Berkeley, 2008.

[57] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *MEMOCODE*, pages 43–50, 2011.

[58] Wenchao Li, Dorsa Sadigh, S. Shankar Sastry, and Sanjit A. Seshia. Synthesis for human-in-the-loop control systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 470–484, 2014.

[59] Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M. Murray. Synthesis of reactive switching protocols from temporal logic specifications. *IEEE Trans. Automat. Contr.*, 58(7):1771–1785, 2013.

[60] Scott C. Livingston, Richard M. Murray, and Joel W. Burdick. Backtracking temporal logic synthesis for uncertain environments. In *ICRA*, 2012.

[61] Scott C. Livingston, Pavithra Prabhakar, Alex B. Jose, and Richard M. Murray. Patching task-level robot controllers based on a local mu-calculus formula. In *ICRA*, pages 4588–4595, 2013.

[62] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51. ACM, 2016.

[63] Matthew R. Maly, Morteza Lahijanian, Lydia E. Kavraki, Hadas Kress-Gazit, and Moshe Y. Vardi. Iterative temporal motion planning for hybrid systems in partially unknown environments. In *Proceedings of the international conference on Hybrid systems: computation and control, HSCC*, pages 353–362, 2013.

[64] ROSwiki. Common Messages. `http://wiki.ros.org/common_msgs` [2018].

[65] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. ORB-SLAM: a versatile and accurate monocular SLAM system. *CoRR*, abs/1502.00956, 2015.

[66] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017.

[67] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference*.

[68] E. Olson. Apriltag: A robust and flexible visual fiducial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3400–3407, May 2011.

[69] Adi Omari, Sharon Shoham, and Eran Yahav. Cross-supervised synthesis of web-crawlers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE*, 2016.

[70] Hila Peleg, Shachar Itzhaky, and Sharon Shoham. Abstraction-based interaction model for synthesis. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2018.

[71] Louise Pryor and Gregg Collins. Planning for Contingencies: A Decision-based Approach. *J. Artif. Intell. Res. (JAIR)*, 4:287–339, 1996.

[72] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[73] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M. Murray, and Sanjit A. Seshia. Reactive synthesis from signal temporal logic specifications. In *HSCC*, pages 239–248, 2015.

[74] Vasumathi Raman and Hadas Kress-Gazit. Automated feedback for unachievable high-level robot behaviors. In *ICRA*, pages 5156–5162, 2012.

[75] Vasumathi Raman, Nir Piterman, and Hadas Kress-Gazit. Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In *ICRA*, pages 4075–4081, 2013.

[76] John W. Ratcliff and David E. Metzener. Pattern matching: The gestalt approach. 13(7):46, 47, 59–51, 68–72, July 1988.

[77] E. Ruiz, R. Acua, P. Vlez, and G. Fernndez-Lpez. Hybrid Deliberative Reactive Navigation System for Mobile Robots Using ROS and Fuzzy Logic Control. In *2015 12th Latin American Robotics Symposium and 2015 3rd Brazilian Symposium on Robotics (LARS-SBR)*, pages 67–72, Oct 2015.

[78] Marcel Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *IJCAI*, pages 1039–1046, 1987.

[79] ROSwiki. Navigation Stack. http://wiki.ros.org/navigation [2016].

[80] Ioan A. Sucan and Sachin Chitta. Moveit! [Online]; available: http://moveit.ros.org [2016].

[81] Tarik Tosun, Ross Mead, and Robert Stengel. A general method for kinematic retargeting: Adapting poses between humans and robots. In *ASME International Mechanical Engineering Congress and Exposition*, 2014.

[82] J.R.R. Uijlings, K.E.A. van de Sande, T. Gevers, and A.W.M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 2013.

[83] Roman van der Krogt and Mathijs de Weerdt. Plan Repair as an Extension of Planning. In *(ICAPS)*, 2005.

[84] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017.

[85] Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Automaton-guided controller synthesis for nonlinear systems with temporal logic. In *IROS*, 2013.

[86] Kai Weng Wong, Rüdiger Ehlers, and Hadas Kress-Gazit. Correct High-level Robot Behavior in Environments with Unexpected Events. In *RSS*, 2014.

[87] Kai Weng Wong, Cameron Finucane, and Hadas Kress-Gazit. Provably-correct robot control with ltlmop, OMPL and ROS. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, 2013.

[88] Kai Weng Wong and Hadas Kress-Gazit. Let's talk: Autonomous conflict resolution for robots carrying out individual high-level tasks in a shared workspace. In *ICRA*, pages 339–345, 2015.

[89] Kai Weng Wong and Hadas Kress-Gazit. From high-level task specification to robot operating system (ROS) implementation. In *First IEEE International Conference on Robotic Computing, IRC 2017, Taichung, Taiwan, April 10-12, 2017*, pages 188–195, 2017.

[90] Kai Weng Wong and Hadas Kress-Gazit. Robot operating system (ros) introspective implementation of high-level task controllers. *Journal of Software Engineering for Robotics (JOSER)*, 2017.

[91] Kai Weng Wong and Hadas Kress-Gazit. Resilient, provably-correct, high-level robot behaviors. ieee transactions on robotics. *IEEE Transaction on Robotics (T-RO)*, 2018.

[92] Kai Weng Wong, Hila Peleg, and Hadas Kress-Gazit. Automatic ros code transfer between robots. *IEEE Robotics and Automation Letters (RA-L)*, 2018. In Submission.

[93] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *HSCC*, pages 101–110, 2010.

[94] Shanchan Wu, Jerry Liu, and Jian Fan. Automatic web content extraction by combination of learning and grouping. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 2015.

[95] O.R. Young. *Bargaining: Formal Theories of Negotiation*. University of Illinois Press, Urbana, IL, 1975.

[96] Meital Zilberstein and Eran Yahav. Leveraging a corpus of natural language descriptions for program similarity. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2016.

[97] Gilad Zlotkin and Jeffrey S. Rosenschein. Negotiation and Task Sharing Among Autonomous Agents in Cooperative Domains. In *IJCAI*, 1989.