Robot Operating System (ROS) Introspective Implementation of High-Level Task Controllers

Kai Weng Wong1,∗ Hadas Kress-Gazit

1 The Sibley School of Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, USA
Emails: {kw358, hadaskg}@cornell.edu

Abstract—In this work, we describe a streamlined process for transforming high-level task specifications into executing programs created with the Robot Operating System (ROS). We leverage both the recent advances in automatic synthesis of correct-by-construction controllers from high-level specifications and the vast availability and functionality of ROS packages. We show how the subscribe-publish paradigm of ROS fits well with the input-output framework of correct-by-construction controllers. With this framework, we automatically detect undesirable behaviors and failures related to the mapping between a correct-by-construction controller and ROS nodes and we automatically provide feedback to the user in the form of suggested changes to the specification when possible faults are detected. We demonstrate our approach with examples involving both single-robot and multi-robot scenarios.

Index Terms—Controller Synthesis, Robot Operating System (ROS), Formal Methods, High-level Control, Implementation Analysis

1 INTRODUCTION

Synthesis of correct-by-construction robot controllers has gained popularity in recent years [1]–[13]; 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 [3] and take into account the complex and nonlinear dynamics of a system operating in a partially unknown environment [4]. 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 [5]. In the case of expected occasional human intervention, we can synthesize a semi-autonomous controller for correct operation [6]. These controllers are also improved to increase robustness against intermittent [8] or chronic [9] unexpected environment events; disturbances in multi-agent systems with both controlled and uncontrolled agents [10]; and exogenous disturbances on a system with continuous dynamics [11]. If controller synthesis fails, we can provide feedback [12] and automatically suggest changes to the task specification [13].

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 1: 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 1 is shown...
shows that with the current specification, the outputs move

denotes that the variable value is False. The yellow node in the middle
automatically synthesized from the specification. The symbol ‘!’

Finite state machine of Example 1. This state machine was
shown in Fig. 1, we abstract the continuous behaviors of the
robot(s) and its(they) environment into Boolean variables –
they are also known as ‘propositions’. In the controller shown
in Fig. 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. 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. 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. 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. 1.
Consider a two-robot scenario: we want to control two KUKA
youBots using the controller in Fig. 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 fo-
cus 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 1 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 execu-
tion and homogeneous robots execution.

Compared with our previous work [16],

1) We removed the constraint of one output proposition
During 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 [9]. Some also look at resynthesizing controllers when unexpected condition occurs during execution [21]. In this work, we provide feedback and suggest changes for any possible anomalies in the system before execution instead.

### 3 Problem Formulation

In this work, we present a system that implements a controller synthesized from a high-level task specification in ROS. We consider manual mapping of correct-by-construction controller inputs and outputs to one or multiple ROS nodes. As described in Section 1, even when executing a correct-by-construction controller on a robot, the system can still create erratic execution. For instance, in Example 1, the robot can move and stop at the same time with the controller in Fig. 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 1:** Given a high-level task specification and a mapping of controller inputs and outputs to ROS nodes, provide:

1. safety and goal-satisfaction guarantees for robot execution with ROS under user assumptions about the environment behaviors in the specification;
2. guarantees on conflict-free message-passing to ROS topics and actions;
3. 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.

### 4 Preliminaries

**Definition 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! [22] 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 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 [1].

In this work, we consider a high-level task specification $\varphi$ 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 [15], we automatically synthesize a robot controller $A$ if the task is feasible. Fig. 1 gives an example of the synthesized controller; it is a finite-state machine. Its edge labels are the input propositions (inputs) $X$ of the controller and its state labels are the output propositions (outputs) $Y$ of the controller.

Propositions are Boolean variables that abstract either the continuous environment behaviors for input propositions $x \in X$ or the continuous robot behaviors for output propositions $y \in Y$. At each state, the controller $A$ first takes in a current valuation of the input propositions. Given the valuation of the input propositions and the current controller state, the controller $A$ 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 5.1) and methods to provide feedback to the user regarding possible problems with the mapping (Section 5.2).

## 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.

### 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. 2 gives an overview of the connection model between a correct-by-construction controller and ROS nodes and Fig. 3 shows the integration of the correct-by-construction robot controller in Fig. 1 with ROS.

In the structure shown in Fig. 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 1, the input person corresponds to the ‘person topic’, as shown in Fig. 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. 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 1, 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.

Similarly, the user also connects each output topic to one node (See Output1 Topic in Fig. 2) or more (See Output2 Topic in Fig. 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. 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 [15] 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 1, 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 '/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 while
the two-robot case is as shown by both orange connections on
the top and bottom of Fig. 3.

The controller node executes the finite-state machine \( \mathcal{A} \)
synthesized from a specification \( \varphi \). 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 be-
tween 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.

5.2 Detecting Possible Failure

With this connection model of a correct-by-construction con-
troller 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 5.2.2)
- **E2.** More than one output proposition publishing to the same
  topic (Section 5.2.3)
- **E3.** An output proposition mapping to multiple nodes and
  some of these nodes are sending commands to the same
  topic (Section 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 5.2.5);
- **C2.** A robot in the group of homogeneous robots is not
  controlled by any output proposition (Section 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. 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 1:
‘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 undesir-
able 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.

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
topic \( t_{prop} \) 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 \( i \)th 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 \( prop \): \( M_{prop} \geq 1 \) and \( i \leq M_{prop} \). We
provide an example below to explain the algorithm to retrieve
all the topics and nodes connected to a proposition.

Consider we start with a ‘proposition node’ \( n_{prop,i} \) con-
Nected 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 through publishing or sending action requests, \( T_{pub}^{prop,i} = \{t_1, t_2, \ldots \} \), and a set of topics reached by a message from
the node \( n_{prop,i} \) through publishing, \( N_{pub}^{prop,i} = \{n_1, n_2, \ldots \} \).
There are no duplicates in \( T_{pub}^{prop,i} \) or \( N_{pub}^{prop,i} \). 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
\( T_{sub}^{prop,i} \) that the ‘proposition node’ \( n_{prop,i} \) is directly or
indirectly subscribing to, a dictionary \( C_{sub}^{prop,i} \) that contains
the paths to different subscribe-reachable topics and nodes,
and a set of nodes visited by the proposition \( prop \) through
publishing, \( N_{sub}^{prop,i} \).

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, \( T_{prop} = \bigcup_{i=1}^{M_{prop}} T_{sub}^{prop,i} \). Note that we do not remove any duplicated elements with this union operation.

With the lists of topics \( T_{sub}^{prop,i} \), nodes \( N_{sub}^{prop,i} \), and the
dictionaries \( C_{sub}^{prop,i} \), where \( a \in \{sub, pub\} \), we can now analyze
the inter-connections of the nodes.

5.2.2 Output proposition nodes publishing to input
proposition nodes

In this integration of ROS with correct-by-construction con-
trollers, we can have output proposition nodes publish mes-
tages 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, \( N_{pub}^{y} = \bigcup_{i=1}^{M_{prop}} n_{pub,i} \),
contains one of the input proposition nodes \( n_x \). If the set
\( N_{pub}^{y} \) contains the input proposition node, i.e.: \( n_x \in N_{pub}^{y} \),
then the algorithm automatically saves the pair of input-
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 \( T_{pub}^{prop} = \bigcup_{i=1}^{M_{prop}} T_{pub}^{prop,i} \)
from Section 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
\( T_{pub}^{y} = \bigcup_{i=1}^{M_{prop}} T_{pub}^{prop,i} \) of one output proposition \( y_p \) with
the set of publishing topics \( T_{pub}^{q} = \bigcup_{i=1}^{M_{prop}} T_{pub}^{prop,i} \) of another output
proposition \( y_q \) for \( q \neq p \). If the intersection of the sets \( T_{pub}^{y} \)
and $T_{\text{pub}}^i$ is not empty (e.g.: $T_{\text{pub}}^i \cap T_{\text{pub}}^j = \{ t, \ldots \}$), then that means both propositions $y_p^i$ and $y_q^j$ 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, $\{(t, p, q), \ldots \}$. 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 1 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 not stop) or (move and stop)’.

### 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 $T_{\text{pub}}^i$ is the union of all the sets of topics $T_{\text{pub}}^i$ of each node $n_{prop,i}$ mapped to the output proposition $prop$. For each set $T_{\text{pub}}^i$ of $n_{prop,i}$, 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 $T_{\text{pub}}^i$ for each output proposition $prop$. If there exists more than one copy of a topic $t$ in the set $T_{\text{pub}}^i$, then we must be from two different sets of $T_{\text{pub}}^i$. 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 $T_{\text{pub}}^i$ and another copy of ‘/youbot_1/cmd_vel’ in the set $T_{\text{pub}}^j$. In the set $T_{\text{pub}}^i = T_{\text{pub}}^i \cup T_{\text{pub}}^j$, there would be two copies of the topic ‘/youbot_1/cmd_vel’: {‘/youbot_1/cmd_vel’, ‘/youbot_1/cmd_vel’, ‘/youbot_1/cmd_vel’, ‘/youbot_1/cmd_vel’, ‘/youbot_1/cmd_vel’}. We can notify the user of this potential undesirable behavior before execution.

### 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 $T_{\text{prop}}^\text{pub}$ i.e., the topics that the output proposition nodes are sending commands to the robots. For all the robot topics ‘/r’ 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 $T_{\text{pub}}^\text{prop}$. 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 1, if the user wants to control two KUKA youBots at the same time when the proposition move is true, but in the set $T_{\text{prop}}^\text{pub}$, 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.

### 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 $T_{\text{prop}}^\text{pub}$ for all output propositions $\gamma$. 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 $T_{\text{prop}}^\text{pub}$, we notify the user.

The reader can find the implementation of our approaches in our ROS package online. A user can create a mapping and

1. [https://github.com/VerifiableRobotics/LTL_stack](https://github.com/VerifiableRobotics/LTL_stack)
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.

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.

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. 4. The specification is as written in Spec. 2.

Example 2: 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 object (line 5-8).

The propositions finished pickup and finished drop in Spec. 2 are known as the completion propositions [23]. 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 [15] 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.

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. 5) into the Plugin. With the list of propositions, the user can either supply an existing mapping file (B in Fig. 5) or create

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. 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. 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 ‘delete’ button right next to each assignment. Once the mapping is done, the user can save the mapping at the end (E in Fig. 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 [25]. 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 [26]. 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! [22]. 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 [27] 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.

### 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 5.2.2 to 5.2.4 to analyze the node connections. The results are given as follow:

**Output proposition nodes publishing to input proposition nodes (Section 5.2.2):**

Fig. 6 gives the result of using the Section 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. 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 5.2.3):**

Both Fig. 7 and Fig. 8 highlight some results of the algorithm described in Section 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. 7, the corresponding nodes of the output region propositions are all publishing velocity commands to the robot through 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 [1], 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 ...).
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. 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 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. 9, we can see that this error is detected and the user can catch this mistake before execution.

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. 12. We use namespaces here to distinguish the homogeneous robots; the namespaces are ‘/sphero_wpw’, ‘/sphero_ggw’, ‘/sphero_rgw’ and ‘/sphero_wpp’.

**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
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. 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 the technique in [12] to analyze and modify the specification. With [12], 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.

Fig. 13: Analysis result of output nodes publishing to the robot velocity topic for Example 3

Before any analysis, first we ask the user to specify all the robots in this task (The Robots row in Fig. 14) and the robot topics that we can ignore in this analysis (The Robots Topics to Ignore row in Fig. 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. 14, we can see that one of the SPRK robots, sphero_wpw is not connected 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. 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.

Fig. 14: Analysis with Section 5.2.5

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

6.3 Discussion

We face some challenges when constructing and using this framework with ROS. 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 2 and currently we cannot detect such a failure.

7 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 considered. For task execution with multiple robots, an analysis on task execution with a group of heterogeneous robots would be useful as well.

REFERENCES


Kai Weng Wong is a Ph.D candidate in Mechanical Engineering at the Sibley School of Mechanical and Aerospace Engineering at Cornell University. Her research focuses on dealing with unexpected events during provably-correct controller execution and extending the use of provably-correct controllers to multi-agent scenarios.

Hadas Kress-Gazit is an Associate Professor at the Sibley School of Mechanical and Aerospace Engineering at Cornell University. She received her Ph.D in Electrical and Systems Engineering from the University of Pennsylvania in 2008 and has been at Cornell since 2009. Her research focuses on formal methods for robotics and automation and more specifically on creating verifiable robot controllers for complex high-level tasks using logic, verification, synthesis, hybrid systems theory and computational linguistics. She received an NSF CAREER award in 2010, a DARPA Young Faculty Award in 2012 and the Fiona Ip Li ‘78 and Donald Li ‘75 Excellence in teaching award in 2013.