From High-level Task Specification to Robot Operating System (ROS) Implementation

Kai Weng Wong and Hadas Kress-Gazit
The Sibley School of Mechanical and Aerospace Engineering
Cornell University
Ithaca, NY, USA
{kw358, hadaskg}@cornell.edu

I. Abstract

In this paper we describe a streamlined process for transforming high-level tasks into executable implementations created with the Robot Operating System (ROS). We leverage both the vast availability and functionalities of ROS packages and recent advances in automatically synthesizing provably-correct controllers from high-level specifications. Specifically, we propose a framework for seamless integration of provably-correct controllers with ROS: we automatically detect possible failures related to the mapping between the controller and the ROS nodes connected to the controller, and we automatically provide feedback to the user in the form of suggested changes to the specification when possible faults are detected.

II. Introduction

Recently, there has been increased interest in automatic synthesis of provably-correct robot controllers from high-level task specifications [1], [2], [9], [20]. These approaches transform high-level specifications detailing what the robot should do into implementations that are provably-correct and that guarantee the high-level task is achieved. While many of these approaches have been demonstrated on physical robots, the low-level integration of the provably-correct, typically symbolic, controller with the hardware is usually a manual and robot-specific ad-hoc process.

Currently, to execute a provably-correct controller on a robot platform, the user manually maps the outputs of the controller to specific programs that execute robot commands; the user also maps the inputs of the controller to other programs that process sensor information. Even though the synthesized controller is provably-correct, the low-level programs that execute robot commands or process sensor information are not and this can lead to failure during execution. Consider the following example:

Example 1: Robot always moves forward. If Robot senses an obstacle, it should stop moving.

Specification 1 Obstacle Sensing Specification

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

For Example 1, the high-level task specification is shown in Spec. 1. This specification is automatically synthesized into a provably-correct controller, in the form of a finite state machine, as shown in Fig. 1. This controller takes in as input the Boolean proposition **person** and sets the values of the output propositions **move** or **stop**, i.e., whether the robot should move and/or stop. There are 4 robot states in total in this controller and they are the circles in Fig. 1. The proposition names in the circles denote the output propositions (if a proposition's value is false, it is denoted with a '!') and the edges are labeled with the input proposition.

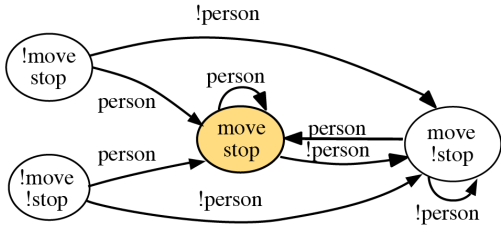


Fig. 1: Finite state machine of Example 1. This state machine was automatically synthesized from the specification.

To execute the controller on a physical system such as a KUKA youBot, we map the outputs **move** and **stop** to a program that executes robot velocity commands such as moving forward and stopping; we also 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 robot controller synthesized from a high-level task specification is provably-correct, the mapping to the lower-level programs that executes the robot commands can create problematic execution. For instance, in Example 1, it is possible to have the robot both move forward and stop at the same time (The center orange state in Fig. 1). This state is valid because there is no restriction on having both **move** and **stop** be true at the same time in the high-level task specification. This results in unpredictable execution as different commands are being sent to the robot at the same time.

Problems similar to the one above, however, are in general hard to detect as the low-level programs connected to the controller can each have a different program structure and format. In this work we restrict the low-level programs to

978-1-5090-6724-4/17 $31.00 © 2017 IEEE
DOI 10.1109/IRC.2017.18

using a similar structure, that of the increasingly popular Robot Operating System (ROS) [14], making it possible to provide feedback to the user about possible failures and even suggest modification to the specification.

The Robot Operating System (ROS) is a growing open-source software library and tool that establishes both a hardware and software standard to work with robots. Because all the components in ROS follow a standard format, it allows people to easily obtain and use the other software packages developed in the community.

In this work, we leverage the standardized structure and libraries of ROS to streamline the high-level-task specification-to-robot-execution process. Specifically, we:

1) Propose a framework for seamless integration of provably-correct controllers with ROS. In this framework, we consider the provably-correct controller as a ROS node, and the inputs and outputs of the controller as nodes, each with a topic dedicated to communication 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 the possible fault described in Example 1 where the robot can stop and move at the same time. In addition, we detect possible failure in situations where the outputs of the controller change the inputs of the controller.
3) Automatically provide feedback to the user in the form of suggested changes to the specification.

Related Work

For anomalies during the execution of provably-correct controllers, researchers have monitored and detected violations of higher-level task specifications at runtime [18]. Other work focuses on resynthesizing controllers during controller execution when anomaly occurs [11]. In this work, we provide preventive measures instead in the form of feedback to the user before the controller execution begins.

For executing high-level tasks with finite state machines, ROS has a package called SMACH [4] that allows users to manually create and execute finite state machines. In our work, our software package is similar but different in that it only requires a specification from the user. The package then checks if a controller can be automatically synthesized leveraging the controller synthesizer Slugs [6]. Our package also provides an executor that runs the controller following a framework described later in the paper. Some researchers have connected these provably-correct controllers with ROS, but they cannot analyze the connected ROS system easily as their programs do not follow the standardized structure of ROS [19].

Because ROS has a standardized structure, researchers can provide guarantees on ROS message delivery between ROS nodes [12]. Other researchers have also created a monitoring infrastructure that intercepts and monitors the ROS messages passing through the system at runtime [8]. For our work, we provide guarantees on robot behaviors instead and we monitor the execution at the task level.

To reduce software development time with ROS, in [7], the authors generated code templates of ROS that developers can implement later on. We are not generating or synthesizing code in this work.

For resilience in robot execution, the authors in [10] implemented an adaptive fault tolerance mechanism for ROS and discussed its feasibility and issues. Researchers in [15] proposed a hybrid navigation architecture with ROS that consists of a high-level deliberative planner with a reactive low-level control and they use Fuzzy Logic to coordinate the two layers. In this work, we propose precautions before robot execution instead.

The rest of the paper is as follows: first we define the problem addressed in this paper (Section III). In Section IV we define the necessary preliminaries. In Section V, we describe our framework and how we can detect and provide feedback to the user; we then illustrate and discuss our approach using an example in Section VI. In Section VII, we summarize our work and discuss future directions.

III. Problem Formulation

In this work, we present a streamlined process to go from a high-level task specification to a ROS implementation. We consider manual mapping of provably-correct controller inputs and outputs to ROS nodes. As described in Section II, even when executing a robot with a provably-correct controller, the system can still create erratic execution. For instance, in Example 1, the robot may be commanded to move and stop at the same time. The resulting behavior is unknown and the robot can crash into a person. We consider the following problem:

Problem 1: Given a high-level task specification and ROS nodes, provide:

(a) guarantees for robot execution with ROS;
(b) 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) Each ROS node does not launch another node during execution.

IV. preliminaries

Definition 4.1: **Robot Operating System (ROS)** The Robot Operating System (ROS) is comprised of packages developed and shared by researchers and hobbyists around the world. ROS operates as a distributed system; to use ROS, users create stand-alone programs called nodes, each denoted as n, with custom or existing ROS packages. Each node n can execute robot commands, update sensor information or process and forward incoming data. 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 is a message bus that can only pass one type of message and it creates a many-in-to-many-out relationship. A node that subscribes to messages from a topic is called the topic's subscriber while a node that publishes messages to a topic is called the topic's publisher.
2) **Services:** Services provide a request/reply relationship in ROS. A node n can send service requests to a service-providing node and wait for replies. The time from sending a request to receiving a reply is usually relatively short. For example, in the MoveIt! [17] 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 it takes is usually less than a second.
3) **Actions:** Actions can be thought of as services that take longer time to fulfill the request. When an action server node receives a request, the server processes the request while in the meantime providing feedback to the action client node. After the action server finishes the request, similar to service, it notifies the client of whether the action request is successful or not.
   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 and action server node both subscribes and publishes to the 'action_topic'.

A user can examine the full system graph, the ROS Computation Graph, $G = \{N, E\}$ of nodes connected to the same ROS master 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 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.

Definition 4.2: **High-level Task specification and Robot Controller Synthesis**

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

Here we give a brief overview of the process to go from a high-level task specification to synthesizing and executing a controller on a robot platform. The reader is referred to [9] for more details.

In this work, we consider high-level task specification φ written in Structured English such as the one as shown in Spec. 1. We can map this specification to Linear Temporal Logic formulas (LTL); using these formulas, we automatically synthesize a robot controller $\mathcal{A}$ in the form of a finite-state machine if the task is feasible [3].

Fig. 1 gives an example of the synthesized controller, which is a finite state machine; its edge labels are the input propositions of the controller and its state labels are the output propositions of the controller.

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 which programs provide inputs to the controller and which programs respond to outputs of the controller.

With the mapping from the inputs and outputs of the controller to the actual robot low-level commands, we are ready to execute the task. In the following section, we describe our framework to connect a synthesized provably-correct robot controller with ROS nodes (Section V-A) and some methods to provide feedback to the user (Section V-B).

V. Approach

Before we can execute the robot controller on a physical system, we need to map each input and output in the controller to a ROS program. In the following section, we elaborate on the framework of controller-to-ROS integration.

A. Mapping from Propositions to ROS Nodes

Since ROS follows a distinct communication paradigm among nodes, instead of asking existing ROS users to learn about provably-correct controller execution, we adapt the execution to the ROS structure. Fig. 2 gives an overview of the connection model between a provably-correct controller and ROS nodes and Fig. 3 shows the integration of the provably-correct robot controller in Fig. 1 with ROS.

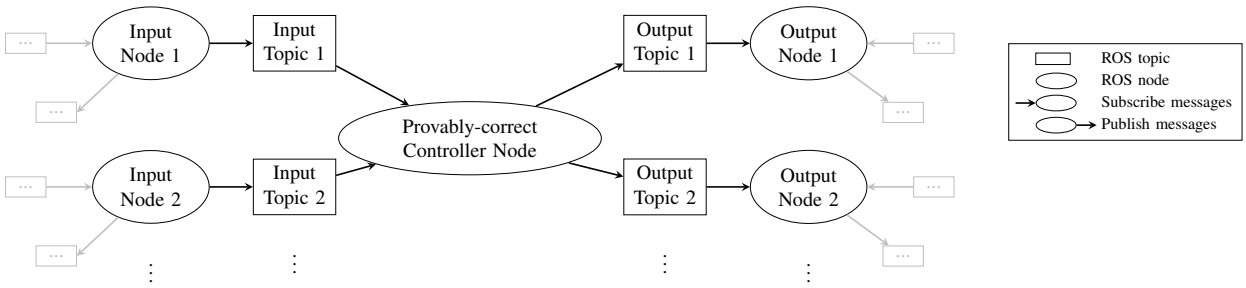


Fig. 2: Overview of ROS Controller Output

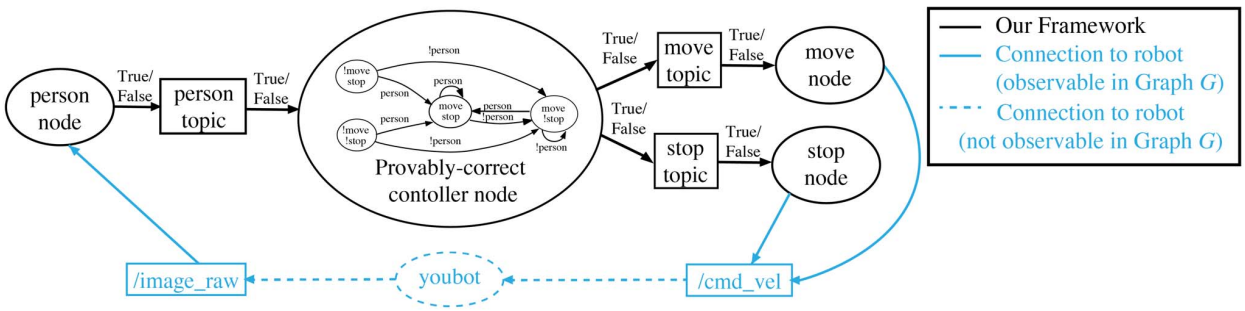


Fig. 3: Integration of the finite state machine in Example 1 with ROS

In the structure shown in Fig. 2, the provably-correct controller forms a standalone ROS node. Each proposition in the controller is mapped to either an input ROS node or an output ROS node. The mapping from a proposition to a ROS node is manually done by the user through a provided GUI. In each of the input nodes and output nodes, there is a dedicated topic, input topic and output topic respectively, to talk to the controller node by passing Boolean messages. In Example 1, the input **person** is the 'person node' and its input topic is the 'person topic', as shown in Fig. 3.

For each input node corresponding to an input proposition, the node first converts sensor information into a Boolean valuation. The node then publishes the value to the controller node through its 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.

For each output node corresponding to an output proposition, the node subscribes to Boolean information from the controller node through its output topic. With the Boolean valuation, the output node then decides on actions to be carried out. 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 **move** receives status messages from the 'move topic' that is published by the controller node. If the status is true, the 'move node' publishes velocity commands to the '/cmd_vel' topic.

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 nodes and publishes the most recent valuation of the output propositions. These output proposition valuations are subscribed by the output nodes through their output topics.

This execution paradigm is now a ROS package and it can be found on *https://github.com/VerifiableRobotics/LTL_stack*. A user can easily create a mapping and execute a controller on a robot platform using our plugin in this package and this is shown later in the Example Section (Section VI).

Currently, when most of the users construct a robot controller with ROS, they create a single or multiple complex nodes that involve both intertwined communications and logical reasoning. The node usually subscribes to and publishes to multiple topics and the node also sends and receives multiple action requests. Since a lot of the conditional reasoning occur inside the single node, it is difficult to analyze the system and detect any possible failures. With our framework, we handle the logical conditions in a provably-correct manner with our synthesized controller. We essentially break down a complex ROS node into multiple simpler ROS nodes and we can then analyze the system.

For the following section, we focus on only 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 for such message-passing method. However, the user can still send and receive service requests and responses in this execution model.

B. Detecting Possible Failure

With this connection model of a provably-correct controller with ROS nodes, we can examine the connection among the nodes. We propose two ways to examine possible undesirable behaviors when executing ROS nodes with a provably-correct controller:

E1. Input propositions subscribing to topics published by output propositions (Section V-B2)

E2. Output propositions publishing to the same topic (Section V-B3)

We have shown in previous sections how E2. can lead to unexpected executions. 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 '/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.

In this section, we show how we can detect these problems before execution and suggest edits to the specification. We refer to both input ROS nodes and output ROS nodes as 'proposition nodes', i.e., nodes that are mapped to a proposition in the specification. Before examining any possible failure, first after the user has mapped all the propositions to a corresponding ROS node, we launch all the 'proposition nodes'. 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 to or publishing to.

1) Retrieve the propositions-to-nodes connections

Publishing or Sending Action Requests: To find out the topics and actions that each 'proposition node' n_{prop} is reaching through publishing or sending action requests, we provide an example below to explain the algorithm.

Consider a case where the 'proposition node' n_{prop} publishes messages to a topic '/A'. Since another node $n_{another}$ 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}, 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 nodes publishes to. The algorithm also ignores nodes that are revisited.

When iterating through all the edges, the algorithm saves i) a dictionary of the paths for the proposition node n_{prop} to reach a topic or node $C^{pub}_{n_{prop}} = \{n_1 : [n_{prop}, t_1, n_1], t_1 : [n_{prop}, t_1], \dots\}$, ii) a list of topics that the 'proposition node' n_{prop} can potentially reach through publishing or sending action requests, $T^{pub}_{n_{prop}} = \{t_1, t_2, \dots\}$, and iii) a set of nodes visited by the proposition $prop$ through publishing, $N^{pub}_{n_{prop}} = \{n_1, n_2, \dots\}$. 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} 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 $T^{sub}_{n_{prop}}$ that the 'proposition node' n_{prop} is directly or indirectly subscribing to, a dictionary $C^{sub}_{n_{prop}}$ that contains the paths to different subscribe-reachable topics and nodes, and a set of nodes visited by the proposition $prop$ through subscribing, $N^{sub}_{n_{prop}}$.

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

2) Output proposition nodes publishing to input proposition nodes

In this integration of ROS with provably-correct 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, $N^{pub}_{n_y}$, contains one of the input proposition nodes n_x. If the set contains the input proposition node, i.e.: $n_x \in N^{pub}_{n_y}$, 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) Output propositions publishing to the same topic

In ROS, we can also 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 lists of publish-reachable topics of each 'proposition node' $T^{pub}_{n_{prop}}$ from Section V-B1, 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 $T^{pub}_{n_{y_p}}$ of one output proposition y_p with the set of publishing topics $T^{pub}_{n_{y_q}}$ of another output proposition y_q for $q \neq p$. If the intersection of the sets $T^{pub}_{n_{y_p}}$ and $T^{pub}_{n_{y_q}}$ is not empty (e.g.: $T^{pub}_{n_{y_p}} \cap T^{pub}_{n_{y_q}} = \{t, \dots\}$), 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 want to notify the user and automatically generate mutual exclusion sentences of these propositions that the user can add into the specification.

First, we automatically save all the pairs of concurrent-topic-access propositions with the corresponding topic, $\{\{t, p, q\}, \dots\}$. Once we 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, when we detect that the output propositions **move** and **stop** both publish to the same topic '/cmd_vel', we can suggest the user to add in a sentence saying that '**move** and **stop** are always mutually exclusive'. In the form of Structured English, it is 'always (not **move** and **stop**) or (**move** and not **stop**) or (not **move** and not **stop**)'.

The reader can find the implementation of our approach in our ROS package online. In the following section, we walk through an example to demonstrate our approach.

VI. Clean and Patrol Example

Consider the following clean and patrol example described in Spec. 2 with a workspace as shown in Fig. 4.

Example 2: Robot patrols all the outer regions, **topLane**, **rightLane**, **bottomLane** and **leftLane**, if it is not holding an object (line 4 in Spec. 2). If Robot sees an object, it will stop and pick up the object (line 1-3). Then Robot heads to **rightGround** and drops off the object (line 5- 8).

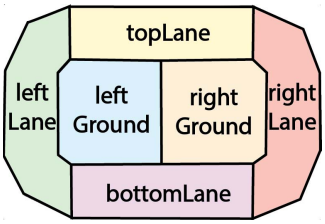


Fig. 4: Map for Example 2

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 specification is feasible and we can automatically synthesize a correct-by-construction controller.

A. *Mapping*

In our online repository, we include a Propositions Mapping and Analysis Plugin and it is GUI that allows the user to easily create mapping from propositions to ROS nodes.

Before mapping propositions to ROS nodes, first the user launches all the nodes to interface with the propositions. Then the user provides a list of propositions for mapping through loading a specification file in the format of .slugsin [6](A in Fig. 5) into the Plugin. With the list of propositions, the user can also supply an existing mapping file (B in Fig. 5) or create a new mapping from scratch.

Based on the nodes connected to the current ROS Master, for each proposition, the user uses the drop down box in the middle (C in Fig. 5) to assign a 'proposition node' to the proposition. Once the user selects a 'proposition node', the user also assigns an input topic or output topic to communicate with the controller node with the drop down box on the right (D in Fig. 5). The user can save the mapping at the end (E in Fig. 5).

For this example, we use a KUKA youBot with one arm and we localize the robot using a Vicon Motion Capture Systems. All the nodes in the ROS structure can subscribe to the robot location leveraging the package vicon_bridge [5]. 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 [16]. We map the **pickup** and **drop** propositions to nodes that plan arm trajectories using MoveIt! [17]. The proposition **pickup** also sends velocity commands to move closer to the object before picking up the object. For

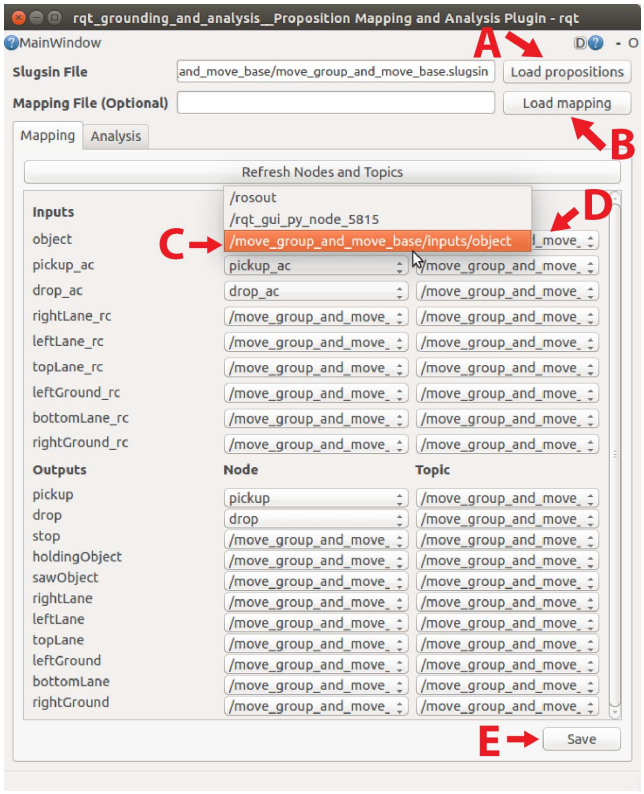


Fig. 5: The Proposition Mapping GUI included in our package. We are mapping propositions to ROS nodes and topics for Example 2 here.

object detection, we use the AprilTag library [13] available in ROS with an RGBD camera mounted in front of the youBot to detect objects. The proposition **stop** sends velocity commands of zero when it is true. With all the propositions mapped to a node, we can now analyze the ROS connections.

B. *Analysis*

Once we obtain a mapping, we can take a snapshot of the current ROS Computation Graph G. With the graph G, we use the algorithms described in Section V-B1 toV-B3 to analyze the node connections. The results are given as below:

1) Output proposition nodes publishing to input proposition nodes

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)

Fig. 6: Analysis result of output propositions publishing to input propositions for Example 2

Fig. 6 gives the result of using the algorithm in Section V-B2. For each row in the table, the leftmost column displays the output proposition that publishes to the input

proposition, the middle column. The path from the output node to the input node in the ROS computation graph G is shown on the rightmost column. 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].

With the current mapping, the **pickup** proposition 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. The algorithm in Section V-B2 provides feedback to the user to ensure the connection is correct.

2) Output propositions publishing to the same topic

Fig. 7: Analysis result of output nodes publishing to the robot velocity topic for Example 2

Fig. 7 and Fig. 8 highlight some results of the algorithm described in Section V-B3. With the youBot, we can control the robot movement by publishing velocity commands to the topic '/cmd_vel'. As shown in Fig. 7, the nodes mapped from 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 them publishes velocity commands at the same time. In our previous work [9], we manually add in the mutual exclusion specification of the region propositions to resolve this issue, but here we can automatically reason about this through the analysis of the ROS Computation Graph.

Besides the output region propositions, we find out the output propositions **stop** and **pickup** are both directly publishing to the velocity topic. Compared with the mutual exclusion of region propositions, the user cannot easily find out this potential conflict before this work.

With this feedback, we suggest the addition of mutually exclusive specification such as '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, as shown in Fig. 8, the **pickup**

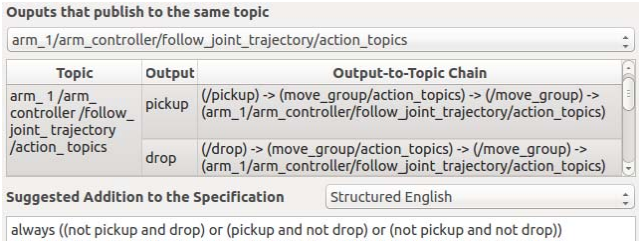


Fig. 8: Analysis result of output nodes publishing to the youBot's arm controller for Example 2

and **drop** propositions can publish to the youBot's arm controller at the same time. In this case, we suggest the addition of '**pickup** and **drop** are always mutually exclusive' to the specification.

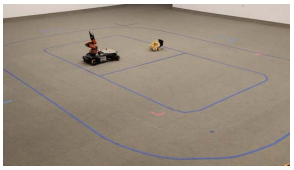
C. Execution

With the analysis and feedback above, the user can modify the specification φ based on the suggestions quickly and launch the controller node with the updated specification. During execution, the robot patrols the outer regions. When the robot senses an object and picks up the object, it heads to **rightGround** and drops off the object. The task continues. None of the mutually excluded propositions publishes to the same topic at any time. The reader can find a video of the analysis together with the robot execution at *https://youtu.be/Wl8Wzk0pqfo*.

(a) The youBot starts in **topLane**.

(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.

Fig. 9: A youBot performing its task as described in Example 2.

D. Challenges

We faced a couple of challenges when developing and using such a framework with ROS.

First, the mapping from propositions to ROS nodes is manually done by the user and this can be cumbersome if there are a lot of propositions.

Also, we spent a substantial amount of time tuning the parameters in ROS packages such as MoveIt! and the navigation

stack to get them working with the youBot. These packages are powerful but they must be customized for different robots.

Lastly, one of the biggest challenges is that when using ROS, the user still need to know about the robot platform. For example, a robot without an arm would not finish the task described in Example 2 and currently we cannot detect such a failure.

VII. Conclusion and Future Work

In this work, we propose a framework for seamless integration of provably-correct controllers with ROS. The subscribe-publish message-passing method of ROS matches with the input-output paradigm of provably-correct controllers. Yet, failure can arise in the low-level execution with ROS when using these provably-correct controllers. In this work, we also describe approaches to detect possible failure and provide feedback to the user using such a system.

In the future, we will expand our work to provide more feedback to the user, such as feedback involving service requests and responses and feedback of failure due the physical constraints of a robot. We want to increase the number of ROS Masters allowed in this framework. We are also considering an integration with SMACH and automating the mapping process in the near future.

Acknowledgments

This work was supported by NSF ExCAPE.

References

[1] 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):61–70, 2007.

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

[3] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

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

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

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

[7] Y. Hua, S. Zander, M. Bordignon, and B. Hein. From AutomationML to ROS: A model-driven approach for software engineering of industrial robotics using ontological reasoning. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Sept 2016.

[8] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. *ROSRV: Runtime Verification for Robots*, pages 247–254. 2014.

[9] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009.

[10] 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.

[11] 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.

[12] Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee. *Verified ROS-Based Deployment of Platform-Independent Control Systems*, pages 248–262. 2015.

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *HSCC*, 2010.