

A Language-Based Approach to Run-Time Assurance for Autonomous Systems

Sumukh Shivakumar

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-99

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-99.html>

May 29, 2020



Copyright © 2020, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to deeply thank my advisor Sanjit A. Seshia, and my mentors Ankush Desai and Hazem Torfah for their immeasurable guidance, support, and patience throughout my research journey. I would also like thank my family for their continuous, unconditional support and encouragement.

A Language-Based Approach to Run-Time Assurance for Autonomous Systems

by

Sumukh Shivakumar

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Spring 2020

A Language-Based Approach to Run-Time Assurance for Autonomous Systems

Sumukh Shivakumar

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

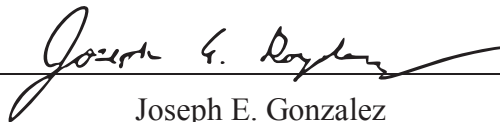


Sanjit A. Seshia
Research Advisor

5/26/2020

(Date)

* * * * *



Joseph E. Gonzalez
Second Reader

May 27, 2020

(Date)

Abstract

A Language-Based Approach to Run-Time Assurance for Autonomous Systems

by

Sumukh Shivakumar

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Distributed Mobile Robotics (DMR) systems are increasingly present in complex autonomous missions, which creates excitement around autonomous robots, but raises questions of safety. Safety of these DMR systems cannot always be guaranteed at design time. To address this, we provide a language-based approach for run-time assurance for robotic systems based on the Robot Operating System (ROS). We present SOTER, an updated run-time assurance (RTA) framework for building safe, distributed robotic systems on ROS. The SOTER framework specifically contains a programming language for implementing reactive robotic software and an integrated run-time assurance system that allows programmers to use uncertified components, but still provide safety guarantees. We demonstrate the efficacy of SOTER using a multi-robot surveillance case study, with multiple run-time assurance modules. Through rigorous simulation, we show that SOTER enabled systems ensure safety, even when using third-party components.

Contents

Contents	i
List of Figures	ii
1 Introduction	1
2 Background	4
2.1 The P Language	4
2.2 Drona Framework	8
2.3 SOTER Framework	11
2.4 Robot Operating System (ROS)	17
3 Run-Time Assurance with SOTER on ROS	19
3.1 Architecture Overview	19
3.2 Language/Systems Contributions for SOTER Integration on ROS	27
3.3 Evaluation	29
4 Conclusion	36
4.1 Future Work	36
Bibliography	38

List of Figures

2.1	DMR Software Stack from [2]	9
2.2	Drona Software Framework from [2]	10
2.3	RTA Architecture	12
2.4	SOTER Semantics	14
2.5	RTA-Protected Motion Primitive from [4]	14
2.6	RTA-Module Declaration	15
2.7	Decision Module Logic	17
3.1	SOTER Framework Architecture	21
3.2	Battery Simulation	31
3.3	Geo-Fence Simulation	32
3.4	Collision Avoidance Simulation	33
3.5	Results of Rigorous Simulation	34

Acknowledgments

I would like to deeply thank my advisor Sanjit A. Seshia, and my mentors Ankush Desai and Hazem Torfah for their immeasurable guidance, support, and patience throughout my research journey. I would also like thank my family for their continuous, unconditional support and encouragement.

Chapter 1

Introduction

There has been an increase in autonomous robots collaborating to perform complex missions in recent times, which has created excitement around the future opportunities offered by distributed mobile robotic (DMR) systems. These robotic systems are commonly involved in diverse and safety critical roles [7], but pose a particularly challenging problem with regards to safety. When designing these DMR systems, safety often cannot be guaranteed unconditionally at design time. These DMR systems also often contain 3rd party components, such as machine learning based components, which are hard to verify for safety. One solution is to use formal verification and systematic testing techniques; however, these techniques are often computationally expensive and have not yet caught up with this increased complexity [21]. There is hence a need to somehow monitor these DMR systems, in order to ensure that they are safe. In particular, there is a need for a systematic way to define monitors for DMR systems so that can be directly integrated with common robot SDKs that are used to build these systems.

These DMR systems have found themselves in numerous use cases, such as package delivery systems, personal transportation, and surveillance. One approach to certifying these systems for safety is to use run-time assurance. This is where the programmer builds a system at design time that has the ability to monitor itself and the surrounding environment at run-time. This system also has the ability to switch to a formally verified safe mode of operation when necessary, even if this means degraded performance.

A popular run-time assurance framework is the Simplex architecture [22]. In the context of robotic systems, this architecture is composed of three main components: (1) the advanced controller (AC) which controls the robot under normal operating conditions, (2) the safe controller (SC) which has been previously certified to keep the robot within a safe region of operation, often at degraded performance, and (3) the decision module (DM) which has also been previously certified to periodically monitor the robot's state and its environment to decide whether to switch from the advanced controller to the safe controller so that it remains with a safe region of operation.

Simplex-based Run-Time Assurance (RTA) frameworks operate with the advanced controller driving the system, paired with the decision module that monitors the overall system

every Δ time to analyze whether the robot is in a state where its safety specification can be violated within Δ time. If it decides that the robot is in imminent danger, it switches control from the advanced controller to the safe controller. These framework are useful, but have limitations in their existing implementations. Many existing techniques [15] [1] [19] apply this framework to just a single component in the entire system, or wrap the entire system using a single Simplex module, making the design of the components of this module often extremely difficult and complicated. Many prior applications also lack programming language support for constructing these RTA systems in a module manner, optimizing only for communication and timing of the system. Some approaches only focus on monitoring capabilities [9] and do not provide a systematic method of switching from the safe controller back to the advanced controller to minimize the overall performance hit. Most importantly, no prior work provides a comprehensive framework with language support for monitor generation and Simplex-based run-time assurance on top of the Robot Operating System. In order to solve these issues, there is a need for a programming framework for building these provably safe robotic systems with run-time assurance that considers all facets of the software stack.

In this work, we build upon SOTER [4], a programming framework used to build safe robotic systems using run-time assurance. The goal of our work is to provide formal monitoring support and run-time assurance for general robot SDK's by extending the SOTER system to become much more modular in supporting a variety of robotics platforms. Specifically, we focus on refactoring the SOTER toolkit to support the Robot Operating System (ROS), which is a common robotics platform. The motivation largely stems from the DARPA Assured Autonomy project to ensure autonomous systems behave correctly as autonomy evolves with machine learning. This effort is in conjunction with Boeing, who have test beds to vet these assured autonomy techniques. These test beds are largely built on top of the ROS platform and so it is clear that having a version of this SOTER run-time assurance software framework on top of the ROS platform would substantially aid in the use of run-time assurance on their automated taxiing project and help advance one avenue of assured autonomy.

In SOTER, a program is built as a collection of nodes, which are periodic processes, that communicate with other nodes in a publish-subscribe mode of communication. An RTA module written in the SOTER language, as in traditional RTA modules, is composed of an advanced controller, a safe controller, and a safety specification which the decision module uses to switch between controllers. The programmer can construct these RTA modules and specify timing behavior for the decision module so the pre-certified safe controller has the ability to switch back to the advanced controller to maximize overall performance of the system. SOTER importantly supports compositional construction of an overall RTA system. In other words, the SOTER contains constructs used to built RTA modules for individual components of the system, and with many such RTA modules, the programmer is able to provide security guarantees for the overall composite system. The SOTER toolkit itself also includes a compiler that generates C code that can be executed on top of the Mavlink software platform.

In this work, we refactor the SOTER framework to support modular components that help port the system on to ROS. Through this process, we make a number of systems contributions in order to allow the SOTER integration on generic SDK's just as ROS. This includes the ability to compile SOTER programs directly as ROS nodes, having P programming language support in ROS's native catkin workspace, and dynamically creating ROS nodes from the SOTER language, directly from application level code in the SOTER program.

In all, we provide a clear software stack for distributed mobile robotics as well as a programming framework for monitoring, which supports a Simplex-based run-time assurance system. This new implementation leverages the original SOTER compiler and generates executable C code that can be directly ported on to robots in addition to simulation platforms such as Gazebo [13] and Open AI gym [12]. We demonstrate the efficacy of this refactored version of the SOTER framework by building a multi-robot surveillance case study. Our results show that this SOTER framework can ensure the safety of multiple robots simultaneously even in the presence of unsafe third-party components and controllers.

In summary, we make the following novel contributions:

1. A refactored programming framework for Simplex-based run-time assurance on top of the ROS platform.
2. SOTER/P language support for native ROS development.
3. A modular robotics software stack to build distributed mobile robotic systems.
4. A sample case study for multi-robot surveillance with multiple RTA modules on different components of the software stack that compose together meeting the overall safety specification of the system.
5. Experimental results in simulation on the ROS platform proving how SOTER can be used for guaranteeing correctness of a system in the presence of third-party unverified components. Videos of our simulations can be found on <https://drona-org.github.io/Drona/> and implementation of our framework can be found at <https://github.com/Drona-Org/Drona>.

Chapter 2

Background

2.1 The P Language

The SOTER framework relies on being an asynchronous event-driven language. SOTER is largely built on top of the P language, which is a domain specific language meant to write asynchronous event driven code easily [3]. It is designed to implement protocols that dictate the interaction between concurrently executing components, which is essential for safe execution of such programs. P also provides the tools in order to be able to create formally safe programs, which are verified through systematic testing. This naturally makes it a good fit for the SOTER language.

A P program is represented as a group of interacting state machines, and each of which communicate asynchronously with events. Events themselves are queued onto a machine's buffer, a first-in-first-out queue, and machines are responsible for handling each of these events in a responsive manner. These events are commonly queued by other machines, but can also be raised from the same machine as well. If a machine does not handle any one of the events that could be potentially enqueued, a failure is thrown during compilation, which is detected during automatic verification.

The *PingPong* P program shown below illustrates the features and semantics of the P language. This program contains 3 machines: *Client*, *Server*, and *Driver*. The goal of this P program is to create a *Client* machine that communicates with two *Server* machines. At a high level, the *Client* machine sends a *Ping* event to both of the *Server* machines sequentially and expects a reply from each one.

An event can be raised by a machine or can be sent from one machine to another. In our *PingPong* example, the *Server* machine raises the *Success* event, which is queued onto its own FIFO buffer from the *SendPong* state. Formally, each state description consists of 4 elements: a state name, a set of events (called the deferred set), a set of event action pairs (called action handlers), and a statement (called the entry statement), which gets executed when a state is entered. A machine contains control states, transitions, actions, and variables. Events cause machines to transition into other states found within the same state

machine. The *Client* machine's *Init* state has an entry statement, where it first instantiates *serverMachines* using payload parameters passed from the *Driver* when creating the *Client* machine. The machine also raises a *Success* event from within the entry block.

```
1 event Ping assert 1: machine;
2 event Pong assert 2: machine;
3 event Success;
4
5 machine Client
6 {
7     var serverMachines: (machine, machine);
8
9     // This is the entry point
10    start state Init {
11        entry (payload : any) {
12            serverMachines = payload as (machine, machine);
13            raise Success;
14        }
15        on Success goto SendPing;
16    }
17
18    state SendPing {
19        entry {
20            send serverMachines.0, Ping, this;
21            send serverMachines.1, Ping, this;
22            raise Success;
23        }
24        on Success goto WaitPong_1;
25    }
26
27    state WaitPong_1 {
28        on Pong goto WaitPong_2;
29    }
30
31    state WaitPong_2 {
32        on Pong goto Done;
33    }
34
35    state Done {}
36 }
37
38 machine Server
39 {
40    start state Init {
41        on Ping goto SendPong;
42    }
43
44    state SendPong {
45        entry (payload : machine) {
46            send payload, Pong, this;
```

```

47         raise Success;
48     }
49     on Success goto End;
50 }
51
52 state End {
53     entry {
54         raise(halt);
55     }
56 }
57 }
58
59 machine Driver
60 {
61     var client : machine;
62     var server_1 : machine;
63     var server_2 : machine;
64
65     start state Init {
66         entry {
67             server_1 = new Server();
68             server_2 = new Server();
69             client = new Client((pongMachine_1, pongMachine_2))
70         }
71     }
72 }

```

Events sent to the machine are stored in the machine's buffer as a FIFO queue and are generally processed in this order by the machine, with one exception. Events themselves can be deferred, which means they can influence the order in which they are delivered to the machine itself. Hence, when the machine is trying to receive an event, it scans its FIFO queue, starting from the beginning and dequeuing the first event that is not in the deferred set. Once an event is dequeued, it is then processed by executing an outgoing transition or by executing an action handler. An action itself is just some code to be executed from within a state. It is the code that is designated to run after a particular event is raised and is entirely dependent on the manner in which the programmer designs the system. In our *PingPong* program, once a *Success* event is enqueued from the entry segment from the *Client* machine, it is then immediately dequeued and transitioned to the *SendPing* state within the same machine.

There are two types of transitions in P: step transitions, and call transitions. In both types of transitions, the transition itself takes the form (n_1, e, n_2) , where n_1 is the source state of the transitions, e is an event name, n_2 is the target state of the transition. Step transitions are the traditional cases, where the *Client* machine raises a *Success* event which then caused it to transition to the *SendPing* state. A call transition has the semantics of pushing the new state on top of the call stack. Call transitions are used to provide a subroutine-like abstraction for machines.

The P language itself has been designed for the implementation of asynchronous re-

sponsive systems, and so it has many built in tools that help the programmer with this implementation. One of the biggest support tools P provides within the compiler itself is checking for unhandled events with automatic verification. If an event arrives in a state and there is no transition defined for this event, then the verifier throws an unhandled event violation. Of course there are certain situations in which the designer of the system would want to delay handling of such events or dropping/ignoring the events all together. In these situations, P allows for these events to be members of the deferred set, and in this case the verifier at compilation time will not flag the events as unhandled. The verifier however also has a built-in liveness check that disallows deferring events indefinitely long. This ensures that the verifier is not simply silenced because every event is in the deferred set.

Under the hood, a P program itself is compiled down from these high level semantics into a “core language” where state descriptions are tuples in the form (n, d, s_1, s_2) , where n is a state name, d is a set of deferred events, s_1 is an entry statement, and s_2 is an exit statement. A P program reduced to its core language consists of event declarations, a nonempty list of machines, and one machine creation statement. Each event declaration also contains a list of types which are sent with an event as payload. Machine declarations now in this core language consist of (1) a machine name, (2) a list of events, (3) a list of variables, (4) a list of actions, (5) a list of states, (6) a list of transitions, and (7) a list of action bindings. Each variable event in the high level P semantics, as seen in the *PingPong* program, has a declared type, and currently P supports int, byte, bool, event, and machine identifier types.

The P compiler takes these P programs and compiles them down to executable C code. The C code is automatically generated with statically allocated data structures, as defined in the P compiler, and these data structures are examined by the P run-time when it executes the operational semantics of the program. The P run-time contains all of the functionality for executing operational P semantics, such as creating machines and enqueueing events on to specific machines. Most of this functionality is kept private to the run-time; however, when integrating another driver framework with the P compiler, often times the run-time needs modifications and foreign functions need to be implemented.

A critical feature of the core language and semantics of the P language is that it allows for the use of foreign functions. This feature is very important in the context of writing real world asynchronous code. To support 3rd party codebases and external code, P allows the programmer to call functions written in the C programming language directly from P programs. These functions are called from within specific P machines and therefore need to be introduced to the machine’s scope with a declaration that gives the function’s name and type signature. The run-time semantics of the foreign functional is nearly identical to a standard C method call. Chapter 3 shows how one must rely on this functionality of P in order to be able to interface with the Robot Operating System’s (ROS) C++ API for the SOTER implementation and experiments.

In all, the goal of the P programming language is to allow the programmer to unify modeling and programming into one activity. Not only can a P program be compiled into executable C code, but it can also be verified using model checking techniques. With its emphasis on asynchronous event-driven programming, P makes a natural choice to use for

SOTER, which relies on many underlying features of P. Our SOTER implementation can be built as an extension to the existing P language, making the necessary changes to the existing P compiler in order to support the run-time assurance language primitives.

2.2 Drona Framework

In order to build this run-time assurance framework for robotic systems, we must also have a reliable framework in which to build and coordinate distributed mobile robotic systems. One of the key components/dependencies to making our SOTER implementation viable on ROS based systems is Drona [2]. Drona is a software framework for building reliable distributed mobile robotics applications. It enables the programmer to program an ensemble of robots with formal guarantees and high assurance of correct operation. Distributed mobile robotics (DMR) systems are those which involve teams of networked robots navigating a physical space to coordinate and accomplish specific tasks. In general distributed mobile robotics systems are becoming increasingly prevalent in complex safety-critical applications [2]. Drona aims to bridge the gap between such systems and providing high assurance and provable guarantees on such systems to enable large scale adaptation. Drona considers the class of DMR systems where the robots are in a known workspace with static obstacles and the tasks to be performed by the robots are generated dynamically. A task represents moving a single robot to a specified goal location.

One of the fundamental problems when working with a team of mobile robotics that share the same workspace is that the programmer must design the system to prevent collisions and still compute optimal motion plans for the individual robots. Additionally, it is difficult to program autonomous reactive robots to properly handle dynamically generated events.

Drona makes the process of building these systems easier by integrating the P language into the Drona tool-chain. As mentioned in section 2.1, P aids the programmer in implementing and specifying asynchronous and event-driven programs. In this case, the Drona software stack can be represented as a series of interacting state machines that communicate with a series of asynchronous events. The P programs are then compiled into C code which can be deployed on a number of robotic systems with the proper interfaces, such as Mavlink [11].

A DMR system implemented using Drona software stack consists of both event-driven asynchronous processes and periodic processes. Hence, Drona has been termed to be a mixed-synchronous system. DMR systems written in Drona can be formally verified using P's built model checking system, Zing. The model checking approach is based on the notion of approximate synchrony where the clocks of each of the robots are not necessarily synced but bounded within a given limit.

The Drona software stack can be divided into 3 main components: the task planner, the software stack, and the robot SDK as displayed in Figure 2.1.

The task planner is where the programmer implements application specific protocols to guarantee that the system satisfies the application specific goals. In the example for a

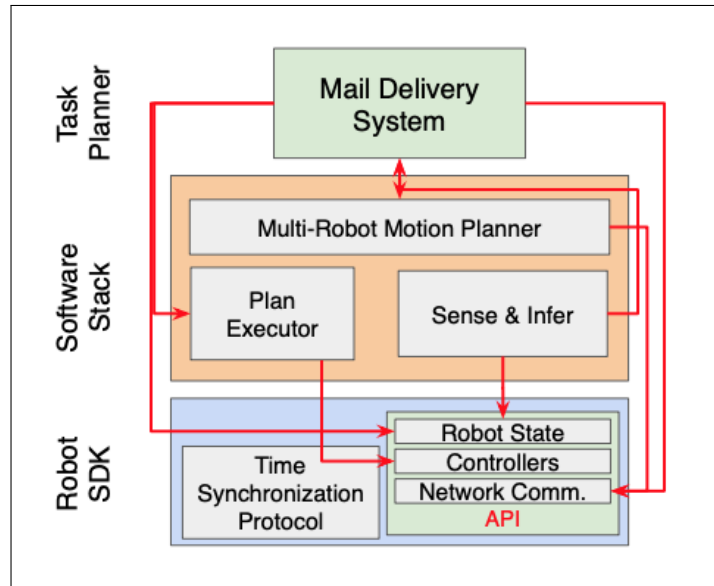


Figure 2.1: DMR Software Stack from [2]

priority mail drone delivery system, the task planner is responsible for ensuring that the mail requests are handled responsively by individual drones and are always delivered in priority order. In an effort to achieve this goal and have a particular drone visit a specific location in the workspace, the task planner sends a request to the motion planner, a separate component inside the DMR software stack. The motion planner is responsible for computing a trajectory to the goal location. The multi-robot motion planner module is responsible for computing safe and collision-free trajectories for each of the robots by coordinating with all other robots within the system. Once the trajectory is computed, the motion planner then sends this trajectory to the plan executor module. It is important to note that the motion planner is a modular component that allows the programmer to swap the multi-robot motion planner with other third-party motion planners, such as the Open Motion Planning Library (OMPL) motion planner [23]. Once the motion planner has computed a path for the robot, it then sends this path back to the Task Planner which forwards it to the Plan Executor module. The Plan Executor module is responsible for interacting with the robot SDK to physically execute the robots' paths in accordance to their designated trajectories. The Plan Executor module ensures that the robot properly follows and executes the path provided by the Motion Planner module.

The Sense and Infer module within the software stack implements monitors that continuously look at a variety of sensor streams from their respective robot and informs the Task Planner if there is something awry and needs immediate attention. The monitors themselves are also represented as state-machines, following P language semantics with asynchronous, event-driven flow of communication between the other Drona stack components. In the ex-

ample of a battery monitor, the P state machine would view inputs streams from the robot’s battery sensor and inform the task-planner when the battery level of the robot is below a certain threshold.

The final component of the overall DMR stack is the Robot SDK. Often times, robot manufacturing companies provide a software development kit that provides robot primitives and functions that allow the programmer to controller robot operations. This SDK also samples the state of the robots and often enables inter-robot communication as well. From a formal verification perspective, the DMR software stack is verified under the assumption that the robot SDK is correct.

Currently, the robot SDK that the Drona software stack support is the Micro Air Vehicle Link (MAVLink) protocol [11]. It is a protocol meant for communicating with small unmanned vehicles. MAVLink is a very lightweight messaging protocol for communicating with drones specifically. It follows a modern publish-subscribe communication model and the point-to-point design pattern. The MAVLink toolchain uses XML message definitions to generate MAVLink libraries. MAVLink, in addition to providing communication between the drones, enables communications between the drone and a ground control station.

The Drona software framework contains four main components: an event-driven programming language for implementing and specifying a DMR application, a reliable DMR software stack, a model checking backend for efficiently verifying the DMR system, and a run-time library for executing the generated C code on a designated robot SDK [2]. The software framework is an extension to the state-machine based programming language P, so that generated C code from the P compiler can be executed on top of the MAVLink protocol. The language is also extended with primitives for specifying workspace configuration. Below is a diagram showing the Drona tool chain.

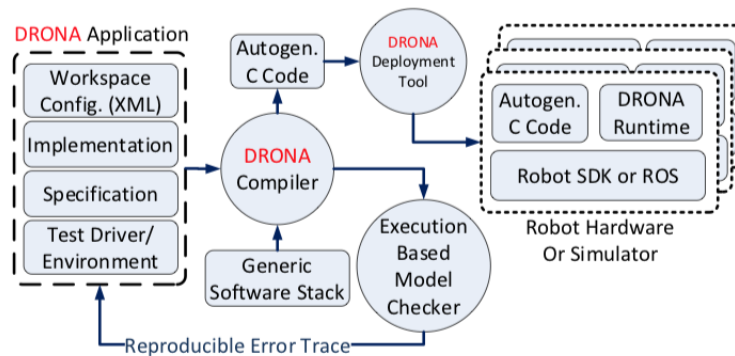


Figure 2.2: Drona Software Framework from [2]

The Drona application, which the programmer implements using the aforementioned extension to the P language, also consists of four subcomponents. The four subcomponents include a workspace configuration XML file, and implementation block, a specification block,

and the test-driver/environment block. The workspace config XML file provides the details regarding workspace information. This information includes the number of robots present in the static workspace, the robot starting positions, locations of all static obstacles, the locations of charging stations for each robot, and any other static workspace related information. Many of the configuration details of the workspace is Task Planner specific, so depending on the application goals, the programmer can add additional relevant information. In this priority mail delivery experiment, designated battery charging stations were added for each of the robots. The specification block is where application specific correctness properties are defined. The specifications themselves are implemented as monitors, or spec monitor machines in the P language. The implementation block is the main collection of P state-machines where the Task Planner is implemented. This is where the programmer designs and develops the Task Planner component from the DMR stack based on application specific goals. Lastly, the test-driver block implements the finite environment state machines that close the DMR system for verification.

The Drona compiler, which is an extension of the P compiler, generates a translation of the application into the Zing modeling language. Zing, in this implementation, has been extended to support mixed-synchronous abstraction to automatically check if the program satisfies the desired properties expressed in the specification block. The compiler also turns the DMR application into C code that is compiled by a standard C compiler and linked against the Drona/P run-time and MAVLink to generate the executable code that can be deployed on to the robot or the simulator.

One of the biggest drawbacks of the vanilla Drona implementation is that it does not natively support robotics systems based on the Robot Operating System (ROS). As described in Chapter 3, one of the main contributions of this research effort has been in migrating Drona to become much more flexible in supporting a variety of other robot SDKs, namely ROS.

2.3 SOTER Framework

Framework Overview

This implementation of SOTER that supports the Robot Operating System, heavily relies on the previous work that comes from the original SOTER paper [4], which provides the theoretical basis for a lot of the work that will be described later. Before that, it is important to understand the theoretical underpinnings that make the design of such a language possible. As previously mentioned, there has been a recent movements towards achieving greater degrees of autonomy and intelligence in robotic systems that also strongly correlates with the complexity of these systems. Especially with the recent drive towards including third-party machine learning components, it is extremely challenging to provide design-time certification, and formal safety guarantees.

One way to minimize the divide between increasingly complex robotic systems and the safety guarantees that can provided is to incorporate techniques of run-time assurance. This

is where the programmer builds a system that is able to monitor itself and the environment at run-time. Using this monitor, the systems should be able to switch to a provably safe operating mode, even if this means degraded performance and sacrificing non-crucial goals. A well-known example of a run-time assurance framework is the Simplex Architecture, which has previously been used for constructing provably-safe avionics [19]. The Simplex architecture is largely based on having 3 main components: (1) the advanced controller (AC) which is responsible for controlling the robot under normal operation, designed for achieving high-performance, but is often not provably-safe; (2) the safe controller (SC) that is pre-certified to keep the robot within a region of safe operation for the robot, but usually at the cost of lower performance; (3) the decision module (DM) which is also pre-certified to periodically monitor the state of the robot and its environment to determine when to switch from the advanced controller to the safe controller so that the system is guaranteed to stay within the safe region. When the AC is in control of the robot, the DM monitors the system's state every Δ period to check if the system can violate the safe specification, and if so, it switches control to the SC.

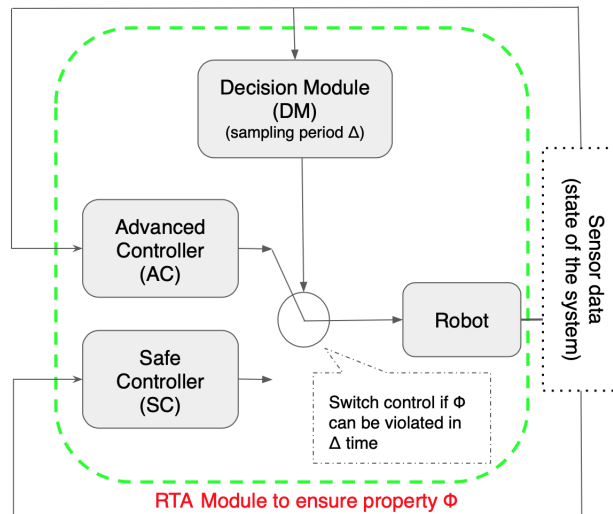


Figure 2.3: RTA Architecture

Desai et. al present SOTER, a programming framework for building safe robotics systems using run-time assurance [4]. In SOTER, a program consists of a collection of periodic processes, called nodes, that interact with each other using a publish-subscribe model of communication. In SOTER, the program consists of at least one run-time assurance (RTA) module. The RTA module consists of an advanced controller node, a safe controller node, and a safety specification. If the programmer defines a well-formed RTA module in SOTER, then the framework guarantees that the system will satisfy the provided safety specification at run-time. The programmer has the ability to declaritively construct an RTA module

with specified timing behavior, combining provably-safe operation with the feature of using the advanced controller whenever safe to achieve good performance. SOTER provides a proper way for the decision module to switch back from the safe controller to the advanced controller, which extends the traditional RTA framework and providing higher performance, since it is in the system’s best interest to maximize AC utilization whenever safe to do so.

As mentioned before, SOTER provides a high-level domain specific language. Additionally, it supports compositional construction of RTA systems. The SOTER language includes constructs that decompose the overall RTA systems into individual RTA modules that are individually monitored and composed in order to provide overall security guarantees for the entire system. SOTER also includes a compiler that generates the decision module node that implements the switching logic. It also generates C code to be directly executed using robot SDKs, such as MAVLink.

When using the SOTER framework, the programmer first needs an application layer to specify objectives for the robotic system. For the purpose of this overview, we can use a robot surveillance system as our application. At the top of the SOTER stack, is the application layer where the programmer specifies system goals, in this case ensures that all surveillance points are visited indefinitely. The application creates target locations for a specific robot. The application layer then communicates with the motion planner to compute a series of way points for the robot to visit that target point. This motion planner communicates these way points sequentially to a motion primitives library, which takes each way point and generates a low level control for the robot to follow the trajectory computed. It is important to note that the underlying dynamics of these controllers, which have usually not been pre-certified for safety, causes deviations from the true reference path. When implementing this application protocol, the programmer often uses many such uncertified components. Examples include third party solvers or libraries like the Open Motion Planning Library [23] to compute way points for the robot. Likewise, motion primitives are themselves often learned using Reinforcement Learning [10], or provided by third-parties without considering safety [8]. In the presence of such uncertified components, it is difficult to provide formal guarantees of safety at design time, especially when the components themselves are hard to verify.

As mentioned, SOTER is a high-level domain specific language based on publish-subscribe model of communication. A SOTER program is a collection of periodic processes that publish and subscribe on a variety of topics, or communication channels. Fig. 2.3 is an example of the SOTER semantics for node creation as well as how nodes publish and subscribe to specific topics.

In the example provided in Figure 2.4, a topic titled `targetWaypoint` is created which is a communication channel that supports messages of type `coord`, which was previously declared as a tuple of floats. This is followed by a node declaration with title `MotionPrimitive` that subscribes to topics `localPosition` and `targetWaypoint`. For the purposes of this example, the `localPosition` topic creation has been omitted. The `MotionPrimitive` node also publishes onto the `controlAction` topic as well. The node runs periodically every 10ms, reads messages from the subscribed topics, performs local computations, and then publishes that control

```

type coord = (x: float, y: float, z: float);
topic targetWayPoint : coord;
fun TTF2D_MPr (s : State): bool {...}
...
node MotionPrimitive period 10;
subscribes localPosition, targetWaypoint;
publishes controlAction;
{
  var currLocation: coord, target: coord;
  /* Get next value at the localPosition and targetWaypoint topic */
  currLocation = localPosition.GetNextValue();
  target = targetWaypoint.GetNextValue();

  /* compute control using third-party SDK */
  ...
  controlAction.publish(control);
}

```

Figure 2.4: SOTER Semantics

action on the controlAction topic.

In the real world, the motion primitives, represented by the motion primitive node, generate control actions to go from the current position to the target position using a third party controller provided by the robot manufacturer. This controller can be considered our advanced, unverified controller in the context of the Simplex architecture mentioned before. These low level controllers often use approximations when modeling the dynamics of the robot, making the simulation performance optimal, not safety optimal. As mentioned already, this is the very same motivation to have a run-time assurance module in the SOTER framework.

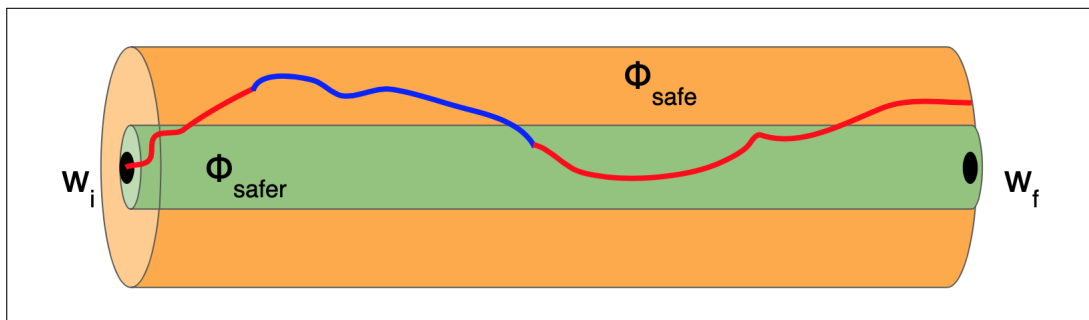


Figure 2.5: RTA-Protected Motion Primitive from [4]

In an example run-time assurance module, as shown in Figure 2.5, the aim is to make the robot go from w_i to w_f , where the former is the current location and the latter is the target location. In this example, the desired safety property is for the robot to always remain inside the region ϕ_{safe} . As seen in the diagram, initially the advanced controller node, which based on our previous example is the MotionPrimitive node, has control over the robot, as shown with the red line. Since the controller is not formally verified it may generate certain control actions that push it outside of ϕ_{safe} . With the run-time assurance module, the decision module then detects this danger and switches to the safe controller, shown with the blue line, in a manner such that there is enough time for the safe controller to gain control over the robot before the robot exits the safe region. This safe controller must be certified to keep the robot inside ϕ_{safe} and also move it to a state in ϕ_{safer} . Once the robot reaches back to this inner green region of ϕ_{safer} , the decision module returns control back to the advanced controller. Figure 2.6 displays the semantics for how to write such a run-time assurance module in the SOTER language.

```

type State = ...;
fun PhiSafer_MPr (s : State): bool {...}
fun TTF2D_MPr (s : State): bool {...}

node MotionPrimitiveSC period 60;
subscribes localPosition, localVelocity, targetWaypoint;
publishes controlAction; { /* body */ }

rtamodule SafeMotionPrimitive =
{MotionPrimitive, MotionPrimitiveSC, 150, PhiSafer_MPr, TTF2D_MPr};

```

Figure 2.6: RTA-Module Declaration

Referencing Figure 2.6, we introduce a new node called the MotionPrimitiveSC as the safe controller node. Here we also have two functions, the first checking if the current state of the robot is in ϕ_{safer} and the second checking whether time to failure, or the time to reach an unsafe state, is less than 2Δ . This function, given a state s , and a predicate $\phi \in SafeStates$, returns true if starting from state s the minimum time after which ϕ may not hold is less than or equal to 2Δ . With all of these components, it is sufficient to build a RTA module declared here as SafeMotionPrimitive, and the decision module then takes care of the switching based on the logic above.

In general, SOTER enables composing multiple RTA modules in order provide larger security guarantees on the over system. These large robotic system are often built by composing multiple smaller components together SOTER is designed to provide RTA modules for these smaller components which can then be composed together to provide system wide run-time assurance. In the robot surveillance example, we want to decompose the stack into

3 components: (1) an RTA-protected motion planner, (2) a battery safety RTA module, and (3) an RTA-protected motion primitive module.

Run-Time Assurance Module

Here, we will formalize the different components of the RTA module.

A topic is formally defined a tuple (e, v) , where $e \in E$ represents the unique name of the topic and E is the universe of all topic names. $v \in V$ is the value that is posted on this topic from V , the universe of of all possible values that can be communicated using topic e .

A node in SOTER can be described as a period input-output state-transition: at every time instant, the node reads the values on its input topics, updates its own state, and publishes values on its output topics. Formally, a node is a tuple (N, I, O, T, C) where (1) $N \in N$ is the unique name for the node from the set of all nodes, (2) $I \subset E$ is the set of all topics this node is subscribed to, (3) $O \subseteq E$ is set of all topics which this node publishes to, (4) $T \subseteq L \times (I \rightarrow V) \times L \times (O \rightarrow V)$ is the transition relation of the node (the node listens to I , transitions its local state and publishes to O), (5) $C = \{(N, t_0), (N, t_1), \dots\}$ is the time-table representing the times t_0, t_1, \dots at which the node N takes a step.

Moving to the run-time assurance module, let S represent the state space of the system, i.e. the set of all possible configurations of the system. The desired safety property is represented as $\phi_{safe} \subseteq S$, meaning the goal is to maintain the robot inside the ϕ_{safe} set.

The RTA module can be formally represented as a tuple $(N_{ac}, N_{sc}, N_{dm}, \Delta, \phi_{safe}, \phi_{safer})$.

1. $N_{ac} \in N$ is the AC node
2. $N_{sc} \in N$ is the SC node
3. $N_{dm} \in N$ is the decision module (DM) node
4. $\Delta \in \mathbb{R}^+$ represents the period of DM
5. $\phi_{safe} \subseteq S$ is desired safety property
6. $\phi_{safer} \subseteq S$ is stronger safety property

Figure 2.7 below, displays the switching logic that sets which controller is in charge of the RTA module given the current state of the system. The DM node evaluates this switching logic once every Δ time units. We can see that if the robot state is in ϕ_{safer} , then it switches from N_{sc} to N_{ac} . The $\text{Reach}_M(s, *, t) \subseteq S$ represents the set of all states reachable in time $[0, t]$ from the current state, using any controller and so the DM checks if the system will remain inside ϕ_{safe} in the next 2Δ time. If not, then N_{sc} must take control until brought back to ϕ_{safer} .


```

if (mode = SC  $\wedge$   $s_t \in \Phi_{\text{safer}}$ ) mode = AC /* switch to AC */
else if (mode = AC  $\wedge$   $\text{Reach}_M(s_t, *, 2\Delta) \not\subseteq \Phi_{\text{safer}}$ ) mode = SC /* switch to SC */
else mode = mode /* No mode switch */

```

Figure 2.7: Decision Module Logic

2.4 Robot Operating System (ROS)

The Robot Operating System (ROS) is an open-source, meta-operating system for robots [18]. Specifically, ROS is a collection of software frameworks for software development of robotic systems. It provides the robotics middleware with services designed for hardware abstraction, low-level device control, message passing between processes, package management, and many others.

When executing, the ROS-based processes are organized as a graph with a network of these processes represented as nodes. The processing takes place in nodes, which receive a variety of messages, including post, control, and planning messages. Between each of these nodes are edges that are represented by high level ROS communication infrastructure. ROS communication also follows a publisher-subscriber model of communication. Each of the nodes publish message data on asynchronous data streams called topics. The robot nodes also subscribe to a variety of these topics in order to learn information from other nodes and react accordingly. This communication model is common amongst robotic systems programming and even in self-driving car architecture [5]. ROS also implements other styles of communication as well such a synchronous RPC-style communication over services. It also supports the data storage on a Parameter Server, which is a shared dictionary that is accessible via network APIs. As a whole, ROS is a distributed framework of several processes that enables different projects/executables to be individually designed. These executables are then loosely coupled at run-time, and often times these processes are grouped into separate packages, which makes sharing and distribution of these nodes much easier. However, an important note to make is that ROS is not a real-time OS, even though there is high importance placed on reactivity and low latency robot control. ROS is capable of integration with real-time code, but because of this lack of native support for real-time systems, the creators of ROS created ROS 2 [17]. ROS 2 is an overhaul of the ROS API, which now takes advantage of more modern libraries and technologies for core ROS functionality and also notably adds support for real-time code and embedded hardware.

The overall ROS ecosystem [18] can be divided into three main subgroups: (1) ROS client library implementations such as roscpp, and rospy, (2) packages containing application-related code which uses one or more ROS client libraries [24], and (3) tools for building and distributing ROS-based software. The ROS client libraries and the language/platform independent tools for building ROS-based software are open source software that are free for commercial use. Most of the third-party packages from the ROS community are licensed

under many open source licenses as well. These packages are often used for common robotic system functionalities such as robot models, perception, planning, simulation tools, hardware drivers, mapping, and many other algorithms.

ROS client libraries are primarily built for Unix systems and officially supports the Ubuntu Linux operating system, because of the open source software dependencies of the libraries themselves. Other operating systems are “supported” by the community, but are largely under experimental development.

The overall goal of the Robot Operating System is to facilitate code reuse in robotics development, and in support of this goal there are a number of features that ROS possesses that make it a very appealing choice for robotics research. Such features include, easy testing with ROS’s builtin unit testing framework, language independence with ROS already having a C++ and Python API, ROS’s thinness making it easy to port and integrate with other frameworks, ROS-agnostic libraries, and scalability, making it appropriate for large development processes. All of these features make ROS such a widely used framework for robotics research as a whole, warranting the shift to ROS for both the Drona and SOTER implementation.

Chapter 3

Run-Time Assurance with SOTER on ROS

This chapter outlines the main contributions of this work, namely the new architecture of the SOTER framework, how it achieves assurance on robotic systems based on the Robot Operating System, and the multi-robot surveillance case study we use to evaluate the efficacy of the framework. The approach we took to designing this framework was to re-design each component and create a narrow interface of communication between each component and the Robot SDK, though separate Cpp modules. Using this framework, the user simply needs to build their program using the SOTER language at the application level, using the components outlined below, and the SOTER framework will automatically create the executable Cpp code that interfaces with the ROS ecosystem. Because we specifically focus on the ROS, we make our framework flexible enough to where any ROS developer can create their ROS packages in their native catkin workspace, but now using SOTER code. This architecture makes it a very modular framework that fits perfectly into traditional robotics development.

3.1 Architecture Overview

The design goal of this new SOTER architecture is to have clearly defined components that abstract different levels of the generic robotics software stack. As mentioned, a huge concern for programming and creating highly reactive robotic systems is to properly handle non-deterministically generated asynchronous events. Hence, we integrate the P state-machine based programming language as the basis for this SOTER framework. Looking at Figure 3.1, the general idea of our architecture is to have the programmer simply implement application logic at a high level using SOTER, and have the rest of our framework use these declarations to generate the necessary components in order to accomplish those application goals. The goal is to have the programmer interact with high-level constructs and declaratively build these robotic systems with monitoring capabilities, and have the framework remove a lot of

the overhead that comes with building these systems traditionally. Hence, we organize the architecture as seen in Figure 3.1, with modular components that create many abstractions between different levels of the framework.

Organizing the various components as P state machines helps simplify the process of implementing and specifying asynchronous event-driven programs. For this reason, the architecture of our refactored version of SOTER on the Robot Operating System borrows from the original Drona DMR stack with some notable differences. These differences include new modules in the Software Stack and a completely different methodology for communicating with the Robot SDK. This new narrow interface of communication to the Robot SDK enables efficient communication, easily monitor-able components, and portability on to other SDKs.

At a high level, looking at Figure 3.1, the programmer primarily interfaces with the Task Planner, where they implement application specific goals. The Task Planner will have one or more robot machines, which are the actual robots used in the application. Each of these robots maps to destinations using their own motion planner, and these plans are executed by the plan executor module. For monitoring, the programmer interfaces with the Sense and Infer module, where they implement monitors that observe certain properties of the application. Lastly each of these components interface the Robot SDK through a thin middleware termed Cpp Modules, which abstract away many of the underlying details needed for ROS programming, and make them accessible in the SOTER language.

The new architecture has been designed such that migrating to new alternatives to ROS in the future incurs minimal overhead, and is flexible to support even ROS 2 [17], LCM [16], MAVLink [11], and ZeroMQ [26], just to name a few. We will outline the new SOTER architecture by first describing each of the components in the framework, as shown in Figure 3.1.

Task Planner

For the purposes of illustrating the different features of the SOTER framework, we will be using a robot surveillance sample case study. We use this case study for later evaluating our framework as well. Starting with the top of the stack, we have the application, which we term the Task Planner. The Task Planner is where the programmer implements all application specific goals. In the context of the robot surveillance example, this application layer implements the robot surveillance protocol that ensures the application specific property, which is to have multiple robots visit a series of different points indefinitely. This is where the programmer includes information on how to instantiate individual robots, what topics/events they're subscribed to, and which points individual robots must visit. The programmer encodes all of this information using the P language. The Task Planner also is in charge of setting up the Robot SDK interface, in this case the ROS interface. We will discuss this more later when discussing the middleware necessary to integrate P into a ROS-based workflow. It is also important to note that the Task Planner is often implemented as a multi-threaded P state machine, so this much of the communication happens

asynchronously, especially in the multi-robot case. In the surveillance protocol, the single instance of the Task Planner is often communicating to two separate robots concurrently in order to have both robots visit locations simultaneously, covering more workspace area in less time for our specific application.

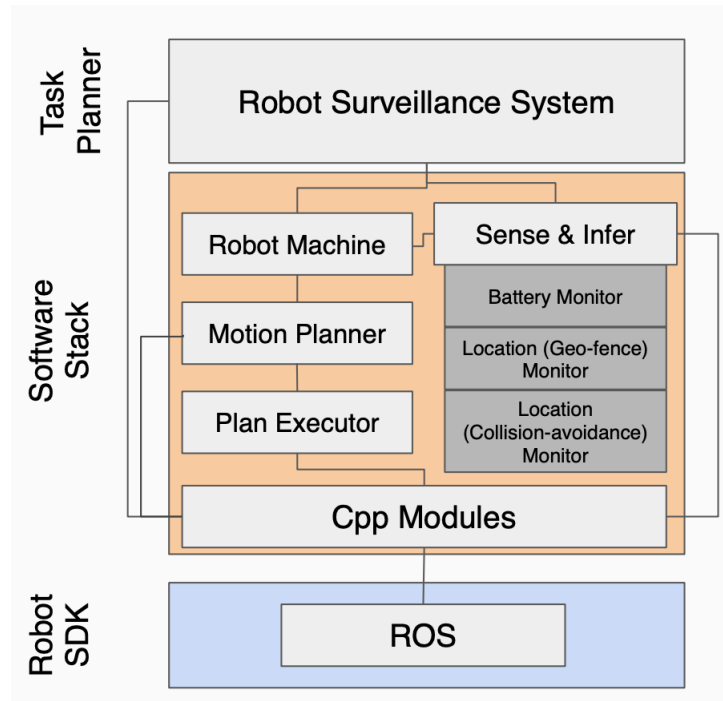


Figure 3.1: SOTER Framework Architecture

Robot State Machine

As shown in Figure 3.1, the next components in the SOTER architecture are the Robot machines. Each of these P based state machines are instantiated from the Task Planner directly and each one represent a different robot as part of the application. Hence, there can be multiple instantiations of the generic robot state machine. The robot machine is responsible for subscribing and listening to events published by the Task Planner for each individual robot, and to act accordingly. In the robot surveillance example, there are two instances of the Robot machine and each robot is responsible for listening to locations sent by the Task Planner. The robot machines also have to initialize themselves with the Robot SDKs, in this case ROS, as soon as they are instantiated by the Task Planner. This is part of the middleware necessary to integrate P into the ROS SDK. The robot machines also need to know the presence of other robots in their workspace, and so the task planner provides

each robot with the information required to learn the positions and communicate with all of the other robots in the application.

Motion Planner

Each Robot state machine creates a designated motion planner state machine. The goal of the motion planner machine is create a path with a series of way points to reach a particular destination in the workspace. With an instantiation for each robot state machine, the motion planner accordingly only computes paths for a specific robot. Hence, when a robot state machine receives a specific destination points from the Task Planner, this information then gets transmitted to the the respective motion planner to compute the corresponding path. The reason for this separation and multiple instances of the motion planner in the stack is again to fulfill the original goal of having the multiple robots survey the workspace concurrently. With this architecture, each robot can compute its own specific paths for its own destinations. The separation of the motion planner from the robot itself allows for a more modular design. This is helpful especially since there are many options for third-party motion planners; this design allows a programmer to swap motion planners easily without having to re-architect their full robotics stack. In the robot surveillance case study, we use the Open Motion Planning Library (OMPL) motion planner [23] in order to compute paths. In our case, we use the OMPL library out of the box, which also relies on a custom workspace parser library to take into account the initial locations of the robots, static obstacles, and other key features of the workspace. This workspace parser reads an XML file to read in this information. This is also particularly important when using multiple motion planners concurrently, which is often the case in the Simplex architecture for run-time assurance. The original SOTER paper has an example with a performant AC motion planner next to a verifiable, safe motion planner, where control is swapped based of the decision module [4]. One thing to note about this motion planner design is that it takes advantage of the P language's ability to be able to call external function written in the C language, through foreign functions as described in Chapter 2. These motion planner machines are hence able to make third party C function calls easily from directly inside the P program, which makes it even easier to integrate third-party motion planners who have C/Cpp APIs. These foreign C functions live in a separate C module that can then be compiled with the P program and executed together. This is the implementation we use in the robot surveillance case study. We will further discuss the compilation process when describing ROS integration with the P language.

Plan Executor

Each of the motion planner machines for their respective robots, creates an instance of a Plan Executor state machine. The Plan Executor machine's goal is to be able to physically execute a path, which is defined by a series of way points in order to reach a destination point, on a specific robot. Hence, each Plan Executor is also uniquely mapped to a specific robot in the

workspace. The Plan Executor listens for paths from its corresponding motion planner, and then interfaces with ROS through foreign functions in order to physically execute the path on to their respective robot. In this case, because ROS only has a Cpp API, we call a Cpp foreign function that gets masked as a C function later down the line with the compiler. More on this interface will be discussed later. Now looking at Figure 3.1, we see a clear, ordered, logical chain of P state machines from the Task Planner, to the Robot State Machines, to their respective Motion Planners, to their respective Plan Executors. These machines are intentionally constructed for events to not only be sent down stream, but also upstream. As mentioned, the overall framework is a multi-threaded application where many aspects of the program will run asynchronously. However, this flow of sending and executing one location at a time to the Robot machine needs to be happen sequentially. Hence, we use thread blocking in P semantics (using the send/receive blocks) to stop further execution of these machines until notified otherwise. More concretely, once the Motion Planner computes its path for a specific target destination, the Motion Planner sends this path to its Plan Executor and then halts all further execution until the Plan Executor finally finishes executing the ROS `goToDestination` function, which only terminates when the robot has finally reached that destination point. The Plan Executor then sends an event to the Motion Planner notifying its completion, and then the Motion Planner also sends this completion message further upstream to the Robot machine to signal it to send the next point that as been queued from the Task Planner. This clearly separated design for the Software Stack enables this bidirectional communication which is essential, and otherwise difficult to implement in robotic systems.

ROS Integration

In order to understand the ROS integration into our P based DMR software stack, we must first overview the traditional ROS development flow using its official build system, catkin [25]. Catkin is a collection of CMake macros and associated code used to build software packages, in this case ROS based packages. To build any ROS-based project, you must first create this project from within a catkin workspace. The catkin workspace is divided into 3 categories: (1) Source Space, (2) Build Space, and (3) Development Space. The source space is where all source code of the catkin package resides. The build space is where CMake is called to build the catkin packages that are found in the source space. CMake is a cross-platform open source tool to manage the build-process of software using a compiler-independent method [14]. It can traditionally be used to build C/Cpp based projects, which is appropriate for ROS Cpp projects. The development space, also termed the devel space, is where the built targets are placed prior to being installed. There is also an inexplicit install space, which does not reside in the workspace. This is where targets are installed, once they are built. Certain packages contain specific nodes, which reside in the source space. These nodes are specific executable programs within the package. Once these packages are built and installed, specific nodes can be run using the `roslaunch package_name node_name` command.

Now before proceeding, we must first overview how traditional P programs are compiled and executed. This will help motivate the integration process with ROS and the catkin workspace. Traditionally, P programs are constructed in either one or more .p files using the semantics described in Chapter 2. These P files are then compiled using the P compiler, which is a .NET Core application. This compiles all of the P programs automatically into C code. In order to actually execute the P program, there needs to be a driver C program with a traditional main method that calls the `PrtMkMachine(...)` function from the P run-time directly on the driving P machine found in the P program. This is to tell the executable which P state machine to start execution from. With this `Main.c` file along with the newly compiled C files as a result of calling the P compiler on the relevant P files, we now have at least 2 C files that need to be built and executed. In P, this done using CMake, so the programmer is responsible for creating a `CMakeLists.txt` file with the relevant files and libraries, which includes all of the P run-time. After running `cmake`, this generates a traditional executable of the P program which can then be run from the command line.

Now having described how both ROS and P operate as standalone systems, we can now describe how to integrate both. We will describe this process in the context of our robot surveillance case study to concretize the steps taken. To start, we have left the ROS build system exactly the same with the traditional catkin flow. The only change in the ROS flow is in the source space for our specific package. We start by creating a stand alone catkin package for our project, which we call `Drona`. This `Drona` package found in the catkin source space then contains separate nodes for different tasks/applications. Within this `Drona` package is where we stray from the traditional ROS development flow. Here we populate our software stack with a series of P files, one for each of the components of the SOTER robotics stack. These P state machines have various responsibilities, some of which are very dependent on the ROS SDK. In these cases, we use foreign function calls from directly in P. These foreign functions live in separate Cpp modules that directly interact with ROS using the `roscpp` API. It is important to note that traditionally P programs only support C foreign functions, so we need to use the `extern "C"` declaration for each of our foreign functions. This makes the function-name in Cpp have C linkage so that the client C code, which will be the P to C compiled programs, can link to these functions using a compatible C header file that contains the declarations of these functions. This is enough to be able to call these Cpp functions from with the P program. Now with the properly formatted foreign functions, the P program is ready to be compiled to C using the traditional dotnet compiler. We also need to make a change to the way we create the driver C program that runs the full P program. Now because the P programs rely on ROS functionality, there is some setup that is required even before calling `PrtMkMachine`, on the first driving P machine. Since again this ROS setup is only possible using the `roscpp` API, this driver program also now needs to be a cpp file with all ROS initialization and then the call to `PrtMkMachine` to initialize the first P machine which in our case is the TaskPlanner machine. We now have all Cpp files containing foreign functions, the compiled C files from our P programs, and the driver Cpp program to initialize the TaskPlanner. We now create the necessary `CMake` files and incorporate all of these files in addition to any external dependencies. In our drone surveillance example, we also needed to

include the P run-time, the OMPL motion planner, and the external workspace parser which is used by the OMPL motion planner. The combination of multiple `CMake` files help declare the executable P program has a ROS executable inside of our `Drona` catkin package. This helps then interface with these executable P programs as any traditional ROS executable and so we can use the `roslaunch` command as before. Even in the traditional ROS development flow, the programmer often has multiple nodes and packages running simultaneously, with each package tasked with a specific application. In our case study, we have the `Drona` package as the executable robot software and we have a separate `multi_robot` package that launches a gazebo simulation of the actual robots used to run our robot surveillance application. This simulation catkin package is the physical workspace to run experiments, and therefore has to have the same features as found in the xml workspace file used as part of our OMPL motion planner.

Overall with this kind of decoupled architecture between the software stack and the Robot SDK, our SOTER implementation of a generic DMR software stack is now much more modular, making ports to other Robot SDKs far simpler. Through the use of foreign functions and `CMake`, we can easily include any generic 3rd party SDK with a C/Cpp API easily.

Sense & Infer Module / Run-Time Assurance

Transitioning into run-time assurance for our SOTER implementation, the final component of our DMR software stack as shown in Figure 3.1 is the Sense and Infer module. The Sense and Infer module is where the programmer is able to include monitors that use a variety of sensors from the robot and observe different properties of the system. Monitors are also responsible for providing run-time assurance based on their observations. Hence, the monitors also implement the decision module logic where they are able to switch control from the advanced controller to the safe controller based on the state of the system. The construction of the monitors themselves is largely dependent on the RTA module and desired safety specification of the application. In general the monitors also interface with many properties of the robot or simulation itself which is all available through the ROS SDK, and hence the Sense and Infer module as a whole also has a dependence on ROS. Like with the previously described components, the Sense and Infer module is entirely implemented as a series of P state machines and because of the ROS dependencies, they again use the Cpp foreign functions from directly within the monitors with a setup similar to that which was described in the ROS Integration.

To aid in our discussion of what goes into the construction of these P based monitors, we will now draw upon the specific run-time assurance modules used in our robot surveillance case study. The first monitor we construct is a Battery monitor that observes battery percentage of the robot. The desired safety property we want for our software stack is to provide battery safety, that prioritizes safely bringing the robot to the charging station when the battery charge falls below a threshold level. Hence, there needs to be a Battery monitor for each of our robots. The monitor itself first gets initialized by the corresponding

Robot state machine with the necessary information to observe the corresponding robot. The battery machines each make foreign Cpp function calls that retrieve the current battery percentage of the robot. We consider the robot to be in a safer state if its battery percentage is above 85%. We design our decision logic, the $\text{ttf}_{2\Delta}$ function, based on the maximum amount battery discharge that could happen in 2Δ time and the maximum charge required to bring the robot to its corresponding charging station. We will go into the specifics of the construction of this RTA in the evaluation section, but given the construction of this function we repeatedly call this function in a loop every Δ time and if the monitor senses that the battery percentage will fall below 85%, then the monitor calls a very specially constructed Cpp foreign function `SwitchACToSC(...)` which switches control from the advanced controller, which receives the current motion plan from the Motion Planner machine and forwards it to the Plan Executor machine, to the safe controller, which is a certified planner that safely brings the robot from its current position to the charging station. Once the robot has safely recharged at the charging station, the decision logic will sense that there is no longer imminent danger and call another foreign function `SwitchSCToAC` to return control back to the advanced controller from the safe controller.

The next P based monitor we use in our robot surveillance case study is a location based monitor in order to geofence our grid-like workspace. In this experiment, our robots travel to a variety of points in a 5 x 5 grid. However, we consider the robot to be in the safer state when in the inner 4 x 4 grid, and safe if in the difference in the outer border, outside of the 4 x 4, but still within the 5 x 5. If the robot is outside of the 5 x 5, then it is in an unsafe state. Hence, in order provide such safety guarantees, we construct a P monitor that exclusively monitors the location of the robot. It is important to note that just like in the battery monitor case, we have a separate monitor for each of the robots, geofencing each independently. The monitor itself follows a similar construction to the battery example, since it also behaves as the decision module. This location based monitor gets instantiated from each of the respective Robot state machines. The monitor then repeatedly makes a foreign cpp function call `MonitorLocation(...)` to observe the robot location. Our Δ for this RTA module is 0.5 seconds and we design our design logic using this value in addition to the current position and the velocities of the robots in order to decide whether the robot will be in an unsafe state within 2Δ . Using this check that happens every Δ time, we again use the same foreign function `SwitchACToSC(...)` to switch to the safe controller when danger is imminent. In this case the Safe Contoller is a certified planner that safely brings the robot back towards the center of our workspace, moving away from the geo fenced barrier as quickly as possible. The monitor still observes the location of the robot and switches back to the advanced controller once back into a safer state using `SwitchSCToAC(...)`.

The third P based monitor we use in our robot surveillance case study is again another location based monitor, but this time to prevent collisions between robots. In this case, we have a separate location monitor that gets spawned, not at the robot level, but at the Task Planner level, where it monitors locations of all the robots simultaneously. In our robot surveillance case study, we have two robots simultaneously navigating the workspace and so we want to make sure that the robots' paths never cross, causing a collision. In this case,

the robot is in a safer state if the robot is more than 0.5 units away from any other robot, and in a safe state if the robot is between 0.3 and 0.5 units away from another robot. If the robot is less than 0.3 units from another robot, then it is considered in an unsafe state. This monitor also acts as the decision module for each of the robots and periodically observes each robot using the `MonitorLocation(...)` foreign Cpp function every Δ time, which is again 0.5 seconds. The decision logic again uses the velocity of each robot in addition to each position to determine if it's in danger. If so, it calls the `SwitchACToSC(...)` to switch to the safe controller which is a certified planner to move the robot back 0.5 units until out of collision territory. Once the robot is back into a safe state, the monitor calls `SwitchSCToAC(...)` to return control to the advanced controller, which is a node that receives the current motion plan from the planner to move towards to the original destination point and forwards it to the Plan executor to execute it using the motion primitives provided by the ROS SDK.

P Interrupt Mechanism

In this architecture, the way the decision modules/monitors switch control from the advanced controller to the safe controller is by interrupting the execution of the Plan Executor module. At this stage, the Plan executor interfaces with the Cpp modules in order to properly execute plans provided by the motion planner. Currently, the Sense and Infer module, which houses the monitors, is indirectly interrupting this flow of execution by switching control at the Cpp module level. However, ideally we need a mechanism that interrupts the flow of execution at the P language level. The monitors implemented in the Sense and Infer modules need to directly interrupt the Plan Executor, rather than interrupting the function calls the Plan Executor makes from the Cpp modules to interface with ROS. This is so that we abstract away as much Cpp code as possible and stick to implementing all flows in this framework using the SOTER language exclusively. Currently, the P language, which we use for implementation, does not support any sort of interrupt mechanism, where event based communication and action handlers can be halted in between. We leave this implementation of an interrupt mechanism in the P language as future work.

3.2 Language/Systems Contributions for SOTER Integration on ROS

This section builds off the ROS Integration subsection, where we describe the key language and systems contributions we make in order to support SOTER/P on top of the ROS platform specifically. This section intends to enumerate the specific contributions and capabilities that are now possible with ROS development using the SOTER framework.

Compiling P Programs as ROS nodes

Using this refactored SOTER framework, developers can now execute programs in the P language as traditional ROS nodes. Programmers are able to create ROS packages just as before, but they can now include P code as part of those packages. Traditionally, ROS developers have a source space where they include all of the application specific code for a specific package. This space can now also include P files, which are then compiled down to executable Cpp ROS executables. These executables can be run just as before with the command `roslaunch ros_package ros_executable`, which can be called from the ROS catkin workspace. Traditionally SOTER/P programs are compiled down to executable C code. In this work, we make modifications to the compilation process of these processes in order to support the traditional catkin compilation process of ROS. These P programs are compiled so that they can be executed natively by the ROS Master node.

Dynamic ROS node generation from the P language

Through this SOTER framework, we now have the ability to dynamically create ROS nodes with their own publishers and ROS callbacks from Task Planner at the P language level. Using this, you can create a Robot machine in P, and all the setup for its publisher, subscriber, and call back information will automatically be handled in the Cpp modules, abstracting away overhead of traditional ROS programming. The publish and subscribe method of communication is used to control the dynamics of the robot, but is something the SOTER developer does not have to worry about as they are only concerned with moving the robot to a series of way points. The callback information is used to provide support for lots of the monitoring capabilities. Again, the P programmer does not need to concern themselves with the details of how the monitor is capturing its information. Rather, they have the ability to simply create monitors that interface with these callbacks. We store all of this P to ROS information in a series of maps, so that there is a one to one correspondence between the P program and the ROS nodes.

As part of this dynamic node generation, we split ROS initialization and all the publish/-subscribe nodes. Traditionally ROS programs have to do some initialization of the master node and all publishers/subscribers at the start of the main program that drives their entire ROS program. We now have many different components in our software stack that are written in P and P programs themselves have a driver C program that initializes the main P machine. In this framework, we decoupled the ROS initialization process between this driver C program, which now had to be modified to a Cpp program, and moved all other ROS initialization into Cpp modules that are called dynamically as new components in P are initialized.

Cpp Integration for P programs

Through this framework, we had to make some modifications to how P programs are compiled so that they can be connected to the ROS ecosystem. One of the main changes we needed to make was to integrate Cpp into P programs, instead of C, since ROS only supports a Cpp API. P programs traditionally are compiled into executable C code, but this is problematic with the ROS Cpp API. This required some changes to the P compilation process, which is now directly integrated with the ROS compilation process. We had to specifically modify the driver C program for P to support Cpp, the functions associated with this driver program (mainly used for making the P program multi-threaded), and all the foreign functions to Cpp, since many had ROS dependencies.

P Integration with third party simulators

Through this framework, in order to actually drive robots using the Robot Operating System, we needed the P code to interface with third party simulators. This is so that the changes in the Task Planner at the SOTER/P level, directly resulted in modifications at the simulation level. Through this refactor, we were able to integrate executable P code with third party simulators. In our case study, we were able to compile and run our P code on a third party gazebo simulator.

3.3 Evaluation

We empirically evaluate the SOTER framework by building an RTA protected robot surveillance software stack that satisfies the following safety invariant: $\phi_{battery} \wedge \phi_{geofence} \wedge \phi_{collision}$. The goal of our evaluation is to show how this new SOTER framework can be used to build a safe software stack for a sample case study, where each component provides security guarantees and satisfies its own safety specification. An important feature of this SOTER architecture is the ability switch from the advanced controller to the safe controller and back, in order to maximize performance. We also empirically evaluate our software stack using rigorous simulation to show that the RTA-protected software stack ensures safety of the system even in the presence of third party components, where the system would have otherwise failed.

SOTER Framework / Experimental Setup

The SOTER framework itself is largely built on top of the state machine based programming language, P. For this reason, it also has the ability to take advantage of many of the tools built into the P language. Specifically, it uses the compiler, the P run-time, and the backend systematic testing engine. The compiler first checks that all components are properly formed and converts the syntax of the P language into executable C code. The P to C run-time then executes this program by using the C representations of the nodes. The periodic behavior

of the processes in the RTA modules are implemented with OS timers and the underlying operating system’s task scheduler. Deploying this case study on a real time operating system is left as future work.

The P compiler also allows the generated C code to be systematically tested by the backend testing engine. This engine enumerates all possible executions of the program in a model checking fashion, controlling the interleaving of nodes using an external scheduler. The third party components, however, are not included and replaced by their abstractions, mainly because the controllers themselves are not implemented by SOTER language, but rather with the middleware used to connect SOTER with the Robot SDK.

The experiments were run on simulation software on two Turtlebot3 robots. We use the latest stable version of ROS, which is ROS Melodic Morenia, on the Linux operating system, specifically the Ubuntu 18.04 distribution. The simulations were conducted in the gazebo simulator environment that has high fidelity models of the Turtlebot3 robot. Videos of our simulations can be found on <https://drona-org.github.io/Drona/> and all implementation of our framework can be found on <https://github.com/Drona-Org/Drona>.

RTA modules

RTA-Protected Battery Safety

The first safety guarantee that we would like to provide for our robot surveillance system is battery safety. We would like to have the stack prioritize safely bringing the robot to its charging station. In the construction of an RTA module for battery safety, we design our advanced controller to be a node that receives the current motion plan for the current destination provided by the Task Planner from the Motion Planner machine and simply forwards it to the Plan Executor machine that then uses the motion primitives provided by the ROS SDK to execute that path. The safe controller is a certified planner that safely brings the robot to its corresponding charging station from the robots current position. We define the robot to be in a safe state in regards to battery safety as long as the battery percentage is greater than 0. We define the safer states for the robot when its battery percentage is above 85%. Because we observe that battery percentage decreases at a relatively slow, we have a relatively large values for our period Δ .

We design our decision logic, the $t_{tf_{2\Delta}}$ function, based on the maximum amount battery discharge that could happen in 2Δ time and the maximum charge required to bring the robot to its charging station. We use the maximum charge required to bring the robot from any point on the workspace as a conservative estimate that is also easy to estimate offline. We define our $t_{tf_{2\Delta}}$ function as the current battery minus maximum discharge that occurs across in time 2Δ . Formally, $t_{tf_{2\Delta}} = b_t - \max_u \text{cost}(u, 2\Delta)$, where b_t is the current battery percentage and the cost function is the amount of battery that robot discharges by applying control, u , for time 2Δ . This guarantees the monitor, and specifically the decision module, switches control to the SC when it anticipates that the robot will not be able to make it to the charging station safely. It also ensures that the DM switches control back to the AC

when the robot is sufficiently charged, which only happens once the robot has sufficiently reached at least 85% charge. Figure 3.2 shows a snapshot from our simulation where robots are approaching their respective charging stations, abandoning their original destinations, when battery is low and the monitor senses that the robots may not have enough charge to reach the destination and arrive at the charging station safely.

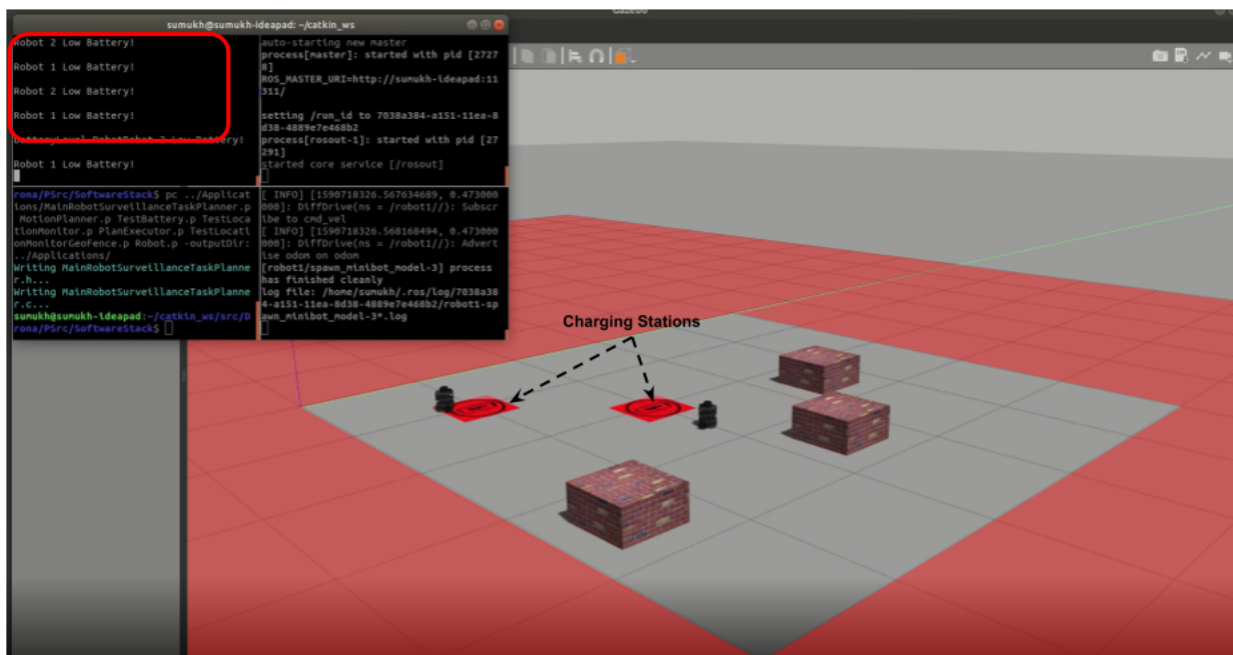


Figure 3.2: Battery Simulation

RTA for Geofencing

The second safety guarantee that we would like to provide for our robot surveillance system is geofencing. We would like the software stack to prioritize safely bringing the robot back within the safe grounds of the workspace. In the construction of this RTA module for geofencing, we design our advanced controller to be a node that receives the current motion plan for the current destination provided by the Task Planner from the motion planner machine that uses the motion primitives provided by the ROS SDK to execute that path. The safe controller is a certified planner that safely brings the robot back towards the center of the workspace, out of the less safe outer boundary of our 5 x 5 grid workspace. We design the robot to be in a safe state as long as the robot has a longitude and latitude between 0 and 5, inclusively. Formally, $0 \leq robot_{id,x} \leq 5 \wedge 0 \leq robot_{id,y} \leq 5$. We define

the safer states for the robot when its in the middle 4 x 4 subgrid within the workspace, or $0.5 \leq robot_{id,x} \leq 4.5 \wedge 0.5 \leq robot_{id,y} \leq 4.5$. Because location of the robot changes quite rapidly we have the smallest Δ period possible of 0.5 seconds, since this is the minimum amount of time needed to refresh the ROS callback in order to read all robots' positions from the monitors.

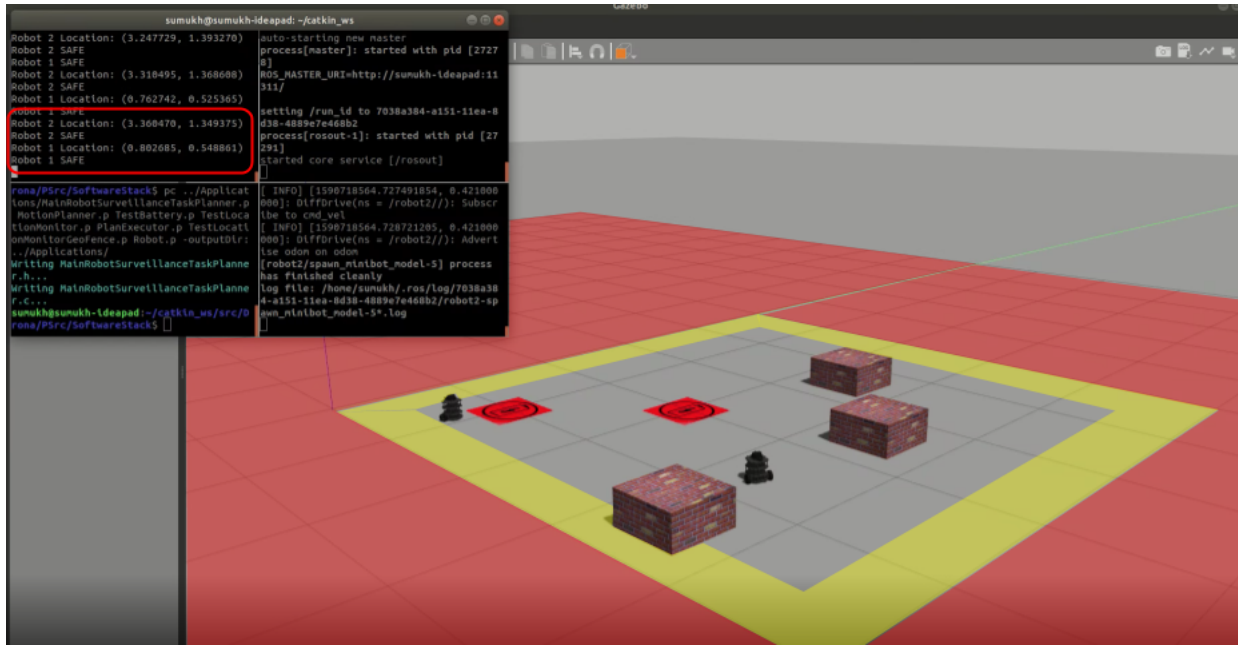


Figure 3.3: Geo-Fence Simulation

We design our $t_{tf2\Delta}$ function based on the maximum amount the robot can move in any direction under any control in 2Δ time. We define our $t_{tf2\Delta}$ as the current position of the robot and the maximum distance change in 2Δ time. This maximum distance is computed under the hood as a function of Δ , as well as the x and y components of the velocity of the robot at time t. Because we have also capped the speeds of the robots themselves, we can easily estimate worst case behavior as a conservative estimate which in turn guarantees that the decisions module switches control to the SC when the robot is in only the safe region, and headed outside of the 5 x 5. In this instant, the SC will gain control and bring the robot from its current position towards the center of the workspace until it reaches the the safer region, in which the AC will gain control of the robot once again, as it attempts to visit the next destination provided by the Task Planner. Figure 3.3 is a snapshot from our simulation and shows the robot moving back to safety and visually depicts the boundaries of what is safer, safe, and unsafe. The robot is attempting to go back to the gray safer zone, as it was originally in the yellow safe zone, but on its way to visit a point in the red unsafe zone.

RTA for Collision Avoidance

The third safety guarantee that we would like to provide for our robot surveillance system is collision avoidance. We would like the software stack to prioritize safely maneuvering out of crashes with other robots in the event where their motion plans cause their paths cross. In the construction of this RTA module for collision avoidance, we design our advanced controller to be a node that receives the current motion plan for the current destination provided by the Task Planner from the Motion Planner machine that uses the motion primitives provided by the ROS SDK to execute that path. The safe controller is a certified planner that safely moves the robot back by 0.5 units radially from the to be collision point. The robot is in a safe state if the robot is more than 0.3 units away from the other robot, or $dist(robot_1, robot_2) \geq 0.3$. The robot is in a safer state if $dist(robot_1, robot_2) \geq 0.5$. Because location of the robot changes quite rapidly we have the smaller Δ period of 0.5 seconds.

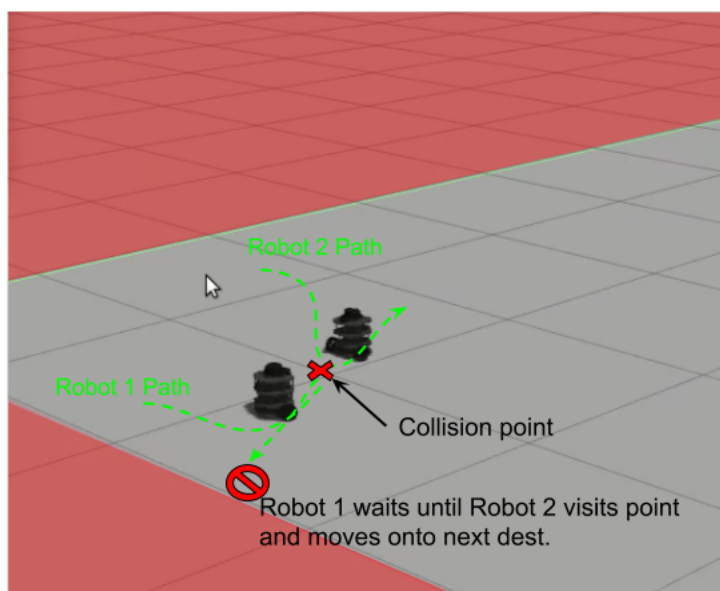


Figure 3.4: Collision Avoidance Simulation

We design our $ttf_{2\Delta}$ function very similar to the geofence version of the $ttf_{2\Delta}$ function. The function is based on the maximum amount the robot can move in any direction under any control in 2Δ time. We define our $ttf_{2\Delta}$ as the current position of the robot and the maximum distance change in 2Δ time. This maximum distance is computed under the hood as a function of Δ , as well as the x and y components of the velocity of the robot at time t. Because we have also capped the speeds of the robots themselves, we can easily estimate

worst case behavior as a conservative estimate which in turn guarantees that the decision module switches control to the SC when the robot is in the safe (and not safer) region, and headed to crash with the other robot. In this instant, the SC will gain control and bring the robot from its current position radially backwards from the collision point until it reaches the the safer region and has one robot wait until the other robot completes its plan, in which the AC will gain control of the robot once again, as it attempts to visit the destination point provided by the Task Planner. Figure 3.4 is a snapshot from our simulation that shows robots have successfully avoided colliding into one another, where both robots move radially out from their original collision point, and one waits as the other completes its plan and moves on to its next destination.

Rigorous Simulation

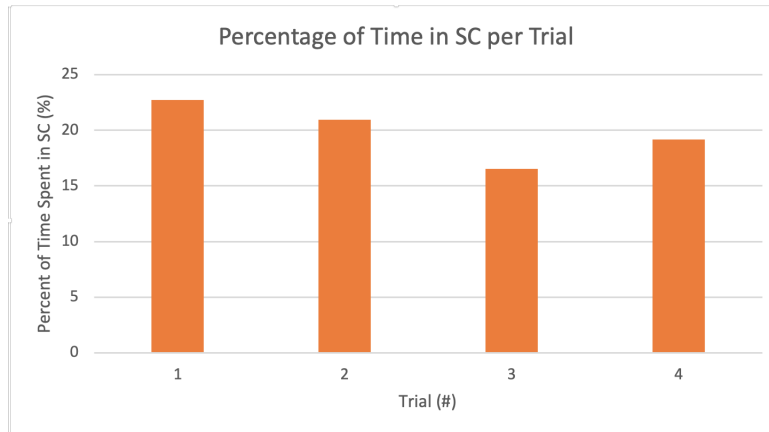


Figure 3.5: Results of Rigorous Simulation

To show that the SOTER frameworks aids in building robust robotics systems with security guarantees, we conducted rigorous stress testing of the RTA-protected robot surveillance software stack. We conducted software in the loop simulations, where the 2 autonomous robots were tasked to visit randomly generated surveillance points in the Gazebo 5 x 5 Grid workspace, while avoiding static obstacles. Totally, the robots drove for approximately 2 hours of simulation, which was plenty given the relatively small size of our workspace the robots were exploring. We conducted this simulation 4 times and we found that on average there were 313 total disengagements, where the advanced controller switched over to the safe controller to avoid a potential safety violation. The majority of these disengagements were due to geo-fencing, where robots would try to explore outside of the boundaries of what is considered safe, and we can attribute this relatively high number again to the relatively small size of the workspace. The recovery times of geo-fencing was relatively small, so it did not hurt overall performance. With all three RTA modules in action, we found that that

several instances of returning control to the AC after recovering the system as well. During our rigorous simulation, the AC nodes were in control on average 80.2% of the time. Thus, safety is ensured without sacrificing overall performance of the robotic system as a whole.

Chapter 4

Conclusion

The focus of this thesis is to provide a language based approach for run-time assurance for robotic systems based on the Robot Operating System. We present SOTER, an updated run-time assurance (RTA) framework for building safe, distributed robotic systems on ROS, but can be generalized to any Robot SDK. In this work, we show that the SOTER framework provides a programming language to implement safe robotic systems using the Simplex architecture. We also provide results showing how to build the decision module that implements the switching logic between the advanced controller and the safe controller, and show its efficacy on a robot surveillance case study. The goal of this thesis is to create a general strategy that can be used to further build such safe systems on ROS or any other SDK.

4.1 Future Work

This work focused on the overall architecture and design of the SOTER framework to be compatible with the Robot Operating System. We focus on a fundamental restructure of the SOTER implementation from its previous iteration and its predecessor Drona, and make it more flexible in supporting a variety of Robot SDKs. One part of this included manually including monitors in the Sense and Infer module that observe specific sensor data streams on the robot in order to decide when to switch from the advanced controller to the safe controller. An interesting new direction would be to automatically generate these monitors in the SOTER/P compiler when given provided an RTA module. The goal would be to have the programmer create an RTA module where they simply define the AC, SC, the safe/safer states, and the $ttd_2\Delta$ function. The compiler should then synthesize the monitor automatically using the RTA language primitives and still function to the standard we present in this work.

An immediate future direction is to implement an interrupt mechanism in the P programming language. Currently, the Sense and Infer module indirectly interrupts the flow of execution and switches between the AC and SC at the Cpp level, where ROS API's are implemented. Ideally, we would like a mechanism that interrupts the flow of execution at

the P language level. This way the monitors can directly interrupt execution and switch control before reaching ROS, abstracting away complexity and leaving logic at the SOTER language level.

Another interesting future direction would be to build a more sophisticated case study. As mentioned, this work was in part motivated by a research initiative from DARPA's Assured Autonomy project in collaboration with Boeing for automated taxiing of their air crafts [6]. With their work progressing in parallel, it would be nice to incorporate their advanced controllers and safe controllers for an automated taxiing case study using this SOTER framework. Ideally, this framework would provide additional security guarantees on top of their current system.

SOTER also provides a nice fit on top other existing work, such as Introspective Environment Modeling [20], where the system can algorithmically generate assumptions on the environment in which the system can operate correctly. Using this technique, SOTER could not only understand when the system is safe, but also automatically synthesize a run-time monitor that can determine when to switch between the AC and SC, a generation process which the publication details. This would thus reduce what the programmer is responsible for when creating their software stack. This would provide a nice end-to-end run-time assurance system that would be very easy to integrate into any robotic system, allowing us to make further progress in the field of verified intelligent autonomous systems.

Bibliography

- [1] Brandon Bohrer et al. “VeriPhy: Verified Controller Executables from Verified Cyber-Physical System Models”. In: *SIGPLAN Not.* 53.4 (June 2018), pp. 617–630. ISSN: 0362-1340. DOI: 10.1145/3296979.3192406. URL: <https://doi.org/10.1145/3296979.3192406>.
- [2] Ankush Desai et al. “DRONA: A Framework for Safe Distributed Mobile Robotics”. In: *Proceedings of the 8th International Conference on Cyber-Physical Systems - ICCPS 17* (2017). DOI: 10.1145/3055004.3055022.
- [3] Ankush Desai et al. “P: Safe Asynchronous Event-Driven Programming”. In: *SIGPLAN Not.* 48.6 (June 2013), pp. 321–332. ISSN: 0362-1340. DOI: 10.1145/2499370.2462184. URL: <https://doi.org/10.1145/2499370.2462184>.
- [4] Ankush Desai et al. “SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019). DOI: 10.1109/dsn.2019.00027.
- [5] *Designing a Connected Vehicle Platform on Cloud IoT Core — Solutions*. URL: <https://cloud.google.com/solutions/designing-connected-vehicle-platform>.
- [6] Daniel J. Fremont et al. *Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI*. 2020. arXiv: 2005.07173 [cs.LG].
- [7] Jérémie Guiochet, Mathilde Machin, and Hélène Waeselynck. “Safety-critical advanced robots: A survey”. In: *Robotics and Autonomous Systems* 94 (2017), pp. 43–52. DOI: 10.1016/j.robot.2017.04.004.
- [8] *Home Page*. July 2019. URL: <https://pixhawk.org/>.
- [9] Jeff Huang et al. “ROSRV: Runtime Verification for Robots”. In: *RV*. 2014.
- [10] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. “Reinforcement Learning: A Survey”. In: *J. Artif. Int. Res.* 4.1 (May 1996), pp. 237–285. ISSN: 1076-9757.
- [11] MAVLink. *Overview*. URL: <https://mavlink.io/en/about/overview.html>.
- [12] OpenAI. *A toolkit for developing and comparing reinforcement learning algorithms*. URL: <https://gym.openai.com/>.
- [13] Osrif. *Why Gazebo?* URL: <http://gazebosim.org/>.

- [14] *Overview*. URL: <https://cmake.org/overview/>.
- [15] Dung Phan et al. “A Component-Based Simplex Architecture for High-Assurance Cyber-Physical Systems”. In: *2017 17th International Conference on Application of Concurrency to System Design (ACSD)* (2017). DOI: 10.1109/acsd.2017.23.
- [16] *Quick links*. URL: <https://lcm-proj.github.io/>.
- [17] *ROS 2 Documentation*. URL: <https://index.ros.org/doc/ros2/>.
- [18] *ROS Wiki*. URL: <http://wiki.ros.org/ROS/Introduction>.
- [19] John D. Schierman et al. “Runtime Assurance Framework Development for Highly Adaptive Flight Control Systems”. In: (Jan. 2015). DOI: 10.21236/ad1010277.
- [20] Sanjit A. Seshia. “Introspective Environment Modeling”. In: *Runtime Verification*. Ed. by Bernd Finkbeiner and Leonardo Mariani. Cham: Springer International Publishing, 2019, pp. 15–26. ISBN: 978-3-030-32079-9.
- [21] Sanjit A. Seshia and Dorsa Sadigh. “Towards Verified Artificial Intelligence”. In: *CoRR* abs/1606.08514 (2016). arXiv: 1606.08514. URL: <http://arxiv.org/abs/1606.08514>.
- [22] Lui Sha. “Using simplicity to control complexity”. In: *IEEE Software* 18.4 (2001), pp. 20–28. DOI: 10.1109/ms.2001.936213.
- [23] Ioan A. Sucan, Mark Moll, and Lydia E. Kavraki. “The Open Motion Planning Library”. In: *IEEE Robotics Automation Magazine* 19.4 (2012), pp. 72–82. DOI: 10.1109/mra.2012.2205651.
- [24] *Wiki*. URL: <http://wiki.ros.org/Client%20Libraries>.
- [25] *Wiki*. URL: http://wiki.ros.org/catkin/conceptual_overview.
- [26] *ZeroMQ*. URL: <https://zeromq.org/>.