# ROSRV: RUNTIME VERIFICATION FOR THE ROBOT OPERATING SYSTEM

BY

CANSU ERDOGAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Grigore Roşu

# Abstract

The Robot Operating System (ROS) is a widely used open-source framework for robot software development. Its increasing popularity, along with its renowned features, such as its dynamic and distributed nature, call for a safety and security protection mechanism which is not supplied as part of the framework. This thesis presents ROSRV, a runtime verification framework for ROS. ROSRV aims to address vulnerabilities in ROS in order to build more reliable robots by enforcing security policies and monitoring safety properties. It integrates with ROS seamlessly; in other words, it does not require any change to the ROS source code or the robot software.

ROSRV has three major components: (1) a tool that provides an expressive formal specification language to define safety properties, and automatically generates monitors out of them, (2) a proxy node that manages these monitors which transparently intercept and observe messages exchanged by the computational units of ROS to ensure the system behaves as desired, and (3) an access control policy administered by the proxy node to restrict the impact of individual units on the overall system.

ROSRV has been tested on a commercial robot running ROS and the evaluations showed promising results.

*To my family with love and gratitude*

# Acknowledgments

I would like to thank my adviser Grigore Roşu, first and foremost, for accepting me as a member of the Formal Systems Laboratory from day one, for all of his rightful complaints when I was disorganized, for his support when I was incapable, for his encouragement and his endless love for what he is doing when I lacked motivation, for his pat on my back whenever I succeeded, and all his other feedback that has helped me get to know myself better, my strengths and my weaknesses. And most importantly, I would like to thank him for giving me the opportunity to be an alumna of UIUC.

Many thanks to everybody who has been a part of my life during this incredible roller coaster, in one way or another, for better or for worse. I am grateful for both the drama and the thrill.

I greatly appreciate all the help I have received, especially from my group members, and would like to thank everyone for their collaboration that has made this thesis possible and much more fun to work on.

Last but not least, thank you Nurgül, Sebahattin and Günsu Erdoğan, for being my rock and keeping me sane. I cannot express how lucky I am to have you three in my life.

This thesis is based on research sponsored by DARPA under agreement number FA8750-12-C-0284. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

# Table of Contents

# Chapter 1

# Introduction

It is the 21st century and robots are becoming more ubiquitous. It goes without saying that they are here to stay due to their capabilities to either replace humans in dangerous duties or to assist them overcome difficult tasks. Current research on robotics is not only on advancing the prevailing human-robot interaction, but it also strives to expedite the ordinariness of fully autonomous robots. By all means, in order for them to be employable, let alone helpful, we need them to operate safely and securely. This creates the need for software that can automatically ensure safe and secure operation of robots.

The focus of this thesis is the ROSRV framework, which is developed to provide Runtime Verification for the Robot Operating System (ROS). ROS owes its increasing popularity to its being an open-source framework supporting many standard operating system services and robot-specific libraries [23]. Its wide adoption calls for a protection mechanism as its current lack of such a feature may pose an important threat against safe and secure operation of robots. Moreover, ROS runs on a heterogeneous computer cluster, and its dynamic and distributed properties make it infeasible to verify the system statically, needless to mention the highly interactive nature of autonomous robots for that matter. Therefore, our approach is to attack this problem with runtime verification. We believe, even without an explicit threat, the fact that the ROS environment is dynamic and versatile with changing parameters and newly joined nodes, justifies that the system benefits from monitoring.

In the rest of this chapter, we will first briefly describe how ROS might be prone to attacks and how monitoring may enhance applications (1.1), then we will talk about our proposed solution and list our contributions (1.2), and lastly we will give an outline of the rest of the thesis (1.3).

## 1.1 Problem Description

In the general sense, the Robot Operating System (ROS) [19] is an open-source framework for developing robot software. ROS operates on top of a heterogeneous distributed cluster of host operating systems and provides a communication layer between nodes which constitute the computational parts of the robot. At runtime, ROS *nodes* (i.e. processes) are connected on a peer-to-peer topology. This particular setup demands a centralized "name service" (referred to as the *ROSMaster* in the rest of the thesis) in order for nodes to find each other and start communicating on channels called *topics*.

After this brief introduction to the overall system, we believe it is important to bring up a few design decisions that play an important role in what ROS does and does not provide.

The primary goal of ROS is to support code reuse across frameworks and applications to help facilitate robotics research [23]. The idea of making it a distributed framework of nodes also increases the feasibility and flexibility of designing individual executables which become loosely-coupled at runtime. However, with the addition of the fact that safety is almost always application-specific into the equation, this greater power brings about greater difficulties in terms of providing a comprehensive methodology as part of ROS to satisfy everybody's safety needs. Since the aim of communication in the system is solely defined by the application, it is more favorable to grant application developers the privilege of specifying safety properties with an easy-to-use and expressive method, rather than to include vague means of safety support as part of the framework which may add undesired complexity.

Another crucial design aspect is the importance of names in ROS. From registering nodes to looking up parameters, everything is associated with a name and has to comply with the naming conventions. There are consequences of giving names the top priority when it comes to identification. For example, when a second node with the same name is introduced to the system, the first node is automatically shut down [10]. This situation makes ROS vulnerable in the case of an attack, as an attacker can easily fake a node and misdirect a robot by publishing bogus messages on important topics. Security issues in ROS is not only limited to taking advantage of this particular design decision. Currently, ROS does not offer any protection

mechanism against preventing nodes from freely querying the ROSMaster about system state and sending shutdown commands to kill arbitrary nodes. These are important problems in need of addressing to build more reliable robots.

## 1.2 Contributions

We developed a runtime verification framework called ROSRV, to improve safety and security of robots developed using ROS. ROSRV integrates with ROS seamlessly in the sense that it does not require any changes to the ROS source code or application code (executed by nodes) itself. Our intention was to (1) address possible safety needs of applications and (2) make up for existing security vulnerabilities in ROS. We approached the first problem by integrating monitoring into the system. This is achieved by placing user-defined monitors as men-in-the-middle in communication channels (i.e. topics) and managing them with the help of a proxy node, called *RVMaster*, located on top of the ROSMaster. The core functionality of monitors is to intercept, observe and optionally modify or drop messages circulating in the system among nodes. We designed our monitors to act like ordinary publishers and subscribers so that the system does not become aware of being monitored. The second problem is avoided by supervising nodes' communications with the ROSMaster. We implemented an access control policy exerted by the RVMaster that dictates which nodes are allowed to send requests and commands to the ROSMaster for execution.

Since our goal is to cater to a variety of applications, our framework had to be easy-to-use and expressive, meaning that users were not to be concerned with the internals of how monitoring works. Therefore, we developed a specification language in compliance with the Monitoring-Oriented Programming paradigm [6], for users to easily specify safety properties. From these specifications, monitors are automatically generated and incorporated into the framework and user-defined actions are executed upon violation or validation of safety properties at runtime according to system behavior.

We tested our framework on LandShark[1], an unmanned ground vehicle (UGV) running ROS, and demonstrated how ROSRV improved the safety

[1]The LandShark UGV is a product of Black-i Robotics (`www.blackirobotics.com`).

and security of LandShark by monitoring the system against specified safety properties and enforcing access control. Our experiments with various monitors showed that our specification language is capable of expressing different kinds of safety requirements, and our framework is successful in delivering user demands.

We also performed performance evaluations based on simple test cases. These tests revealed that no matter how long the execution time, the number of messages not received by the subscriber due to monitoring delay, does not exceed a few.

Our main contributions can be summarized as follows:

1. We developed a simple and expressive specification language and a tool called *ROSMOP* for users to define safety properties without being obliged to know the internals of the ROS communication system for monitoring needs.

2. We implemented ROSRV, a runtime verification framework for ROS, that manages monitors automatically generated by ROSMOP out of specifications, to check dynamic behavior during system execution without the system being aware.

3. We integrated an access control mechanism into our framework to restrict the communication of nodes with the ROSMaster in order to prevent possibly malicious commands to be executed arbitrarily.

## 1.3 Outline of Thesis

The rest of this thesis is organized as follows:

Chapter 2 introduces the Robot Operating System concepts (2.1) and Monitoring-Oriented Programming (2.2) in more detail. Chapter 3 discusses several concerns in general to present our motivation, with a focus on possible shortcomings of ROS on our case study robot LandShark to illustrate the existence of the problem. In Chapter 4, we present our framework ROSRV, by focusing on three major components: RVMaster (4.1) is the proxy node which sits on top of the ROSMaster; it is in charge of regulating monitors (4.2) and administering system accessibility (4.3). In Chapter 5, we present

our evaluation metrics and results. Lastly, we conclude our work and talk about possible future directions in Chapter 6.

# Chapter 2

# Background

In this chapter, we talk about the ROS communication concepts and introduce Monitoring-Oriented Programming.

## 2.1 The Robot Operating System (ROS)

The Robot Operating System (ROS [19]) is an open-source meta-operating system for robot software development. Although it is not an operating system in the traditional sense of process management and scheduling, it provides certain services that an operating system would [23], such as hardware abstraction, low-level device control, various filesystem functionalities, message-passing between processes, etc. Since ROS's main goal is to facilitate code reuse in robotics research and development (including across platforms), it also provides tools and libraries to make the code distribution and execution as easy as possible. The framework is already implemented in at least 3 languages to aid this cause, with a few other languages in currently experimental stage. This thesis focuses on the C++ implementation.

ROS communication is based on a peer-to-peer (potentially distributed) network of processes. Due to these peers becoming loosely-coupled at runtime, this communication infrastructure is called a "graph." There are a few different styles of communication supported by ROS for processes to connect. These are mainly synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

The following basic concepts make up the *ROS Computation Graph* [20] by contributing to data exchange in the framework in one way or another.

**Nodes** Nodes are the previously referred processes that perform computation and help control the robot collectively. Since ROS is designed to be

modular, usually a robot control system consists of many nodes, each of which has a distinct task in terms of computation in order to help lead the robot. ROS nodes are written by using ROS client libraries available for different implementation languages.

**Master** The *ROSMaster* (as referred to in this thesis for clarity) is responsible for providing name registration and lookup. The rest of the components constituting the Computation Graph benefit from and depend on the services the ROSMaster provides to find each other and communicate.

**Parameter Server** The Parameter Server can be considered as a shared dictionary that nodes use to store and retrieve data. It is currently implemented as part of the ROSMaster.

**Messages** Inter-node communication happens by means of messages. A message resembles a C struct in the sense that it is a data structure containing typed fields of either primitive types (e.g. integer, floating point, boolean, etc.), arrays of primitive types, or nested structures.

**Topics** The most common case of inter-node communication in ROS is based on publish/subscribe semantics. In this transport system, a node sends out a message by publishing it to a particular topic, and a node that is interested in a certain sort of data subscribes to the proper topic. In this sense, a topic is simply a name for identifying the content of the message. In ROS, multiple concurrent publishers and subscribers are allowed for a single topic, as well as, a single node publishing and/or subscribing to multiple topics. It is important to note that in this system, publishers and subscribers are not required to know of each others' existence in advance. This design serves the purpose of decoupling the production and consumption of information.

**Services** Services are used for request/reply interactions where many-to-many publish/subscribe semantics falls short. For this type of transport system, a pair of message structures -one for the request and one for the reply- is needed. This type of communication is usually similar to a remote procedure call, where a node offers a service associated with

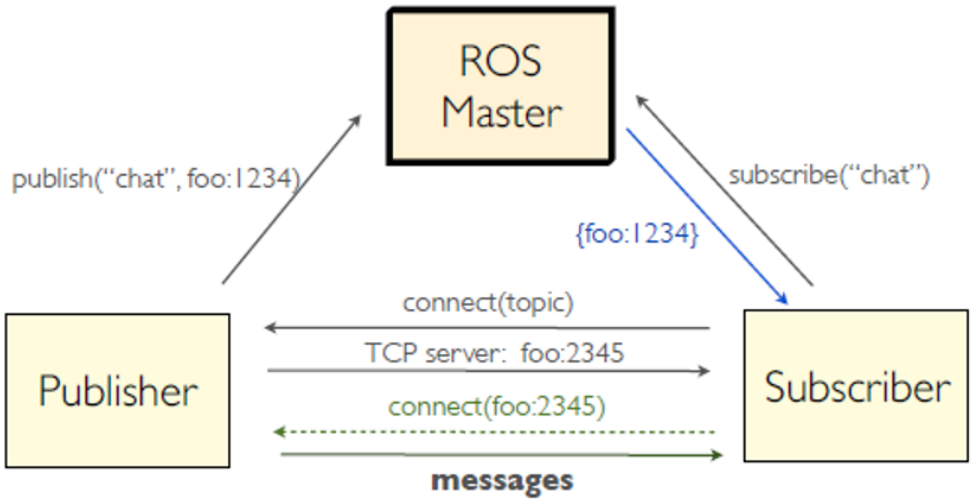


Figure 2.1: ROS Communication Architecture

a name, and another uses the service by sending the request message and awaiting the reply.

**Bags** Bags are helpful in saving and replaying ROS messages. They especially come in handy for storage and analysis of data, such as messages emitted by a sensor on the robot, which might be crucial in developing and testing algorithms.

Figure 2.1 illustrates an overview of the ROS communication architecture. ROS nodes run XMLRPC servers managed by the ROS client library, and each node is assigned a *URI*, which corresponds to the host:port of the XMLRPC server they are running. ROS nodes use XMLRPC connections for two purposes. First, they need to report their registration information to the ROSMaster. This XMLRPC call includes the node's URI, the name of the topic of interest, the data type (i.e. message name) for the topic, and a parameter for the ROSMaster to be able to make callbacks to the node in case new publishers/subscribers join the system wanting to connect on the same topic or when a node's registration information has changed. ROS client libraries also support commands that query/update the system parameters and the runtime state by communicating with the ROSMaster, such as asking for published/subscribed topics of a node, killing a node, getting a list of all the published topics and their types, etc. The second use of XMLRPC by nodes is for peer-to-peer connection negotiation and configuration. However,

this type of communication is not used to transport topic or service data.

As the figure indicates, nodes connect to other nodes directly. The role of the ROSMaster in inter-node communication is to only provide the lookup service for subscribers to find available publishers on particular topics via callbacks. The use of callbacks and the important role given to names ease the decoupling of the two sides. Publishers may generate messages on topics without actually knowing if there are any subscribers listening. And subscribers may show interest in listening to topics without a publisher being present. Eventually for a connection to be established and messages to be routed, the essential thing is that the topic and the declared type (i.e. message) should be matching, which both come down to names. This allows the two parties (i.e. publishers and subscribers) being started, killed, and restarted in any order without causing any errors.

The following is a likely scenario that explains the initiation of message exchange between nodes [24]:

1. Subscriber starts and registers with the ROSMaster with the information on which topic it wants to subscribe to (*via XMLRPC*)

2. Publisher starts and registers with the ROSMaster with the information on which topic it wants to publish to (*via XMLRPC*)

3. The ROSMaster informs Subscriber that a new Publisher has joined via a callback and passes on Publisher's URI (*via XMLRPC*)

4. Subscriber contacts Publisher to request a topic connection and negotiate the transport protocol (*via XMLRPC*)

5. Publisher sends Subscriber the settings for the selected transport protocol (*via XMLRPC*)

6. Subscriber initiates the connection with Publisher using the selected transport protocol (*via TCPROS, etc.*)

For transport protocol negotiation in the 4th step, Subscriber sends Publisher a list of supported protocols. Publisher then selects an appropriate protocol from that list in the 5th step, and returns the necessary information back to Subscriber so it can establish a separate connection for message transfer. The most general protocol used in ROS for topic transport is TCPROS,

which uses stateful, persistent TCP/IP socket connections. UDPROS is another transport protocol supported by ROS.

## 2.2 Monitoring-Oriented Programming (MOP)

Monitoring-Oriented Programming (MOP) [6] is a generic monitoring framework which aims to reduce the gap between formal specification and implementation by comparing the latter's accuracy against the former at runtime. MOP's main goal is to support and encourage building reliable software. MOP tools provide automatic synthesis of monitors out of user-specified properties defined using logical formalisms, and their integration into the original system. These integrated monitors observe the system's dynamic behavior during execution, and trigger user-defined handling actions upon validation or violation of a property. MOP can be classified as (1) a lightweight formal method, (2) an extension to programming languages with logics, and (3) a discipline allowing one to improve safety, reliability and dependability of a system by monitoring its requirements against its implementation at runtime. It is important to note that despite being firmly based on logical formalisms and mathematical techniques, MOP is not an attempt on program verification. The philosophy behind MOP is to exactly not verify an implementation against its specification before operation, but to not let it go wrong at runtime.

MOP *instances* are named after the programming language or platform they are developed for. So far there are three MOP instances which are JavaMOP [5], BusMOP [2], and ROSMOP [8], the last being the focus of this thesis. All MOP instances share the following five orthogonal attributes of the MOP framework [16]: programming language, logic, scope, running mode and handlers. The programming language decides the language of the applications that are to be monitored. In the case of ROSMOP, this corresponds to C++, as we deal with the C++-implementation of ROS. The logic refers to the formalism used to specify the property. MOP offers a selection of logics so that users can choose the most appropriate one for their applications, or omit it altogether if that is the best fit. ROSMOP currently supports FSM and CFG logics. The scope specifies where to check the property. Its

variety may change from instance to instance. The running mode denotes where/when the monitoring code runs. ROSMOP monitors are integrated into the ROSRV framework and they are run *online*, i.e. simultaneously with the other publishers and subscribers of the topic of interest. The handlers specify what actions to be taken upon violation or validation of properties. Even if users choose not to define properties via formalisms for various reasons, MOP still provides means for writing custom monitoring codes and their execution. The rest of this chapter focuses on the MOP framework in general; more information on ROSMOP can be found in Chapter 4.2.1.

Every MOP instance extends the MOP framework in four dimensions: (1) a specification language based on the problem domain which prescribes how to define events, (2) a target language for generated monitors, (3) supported logic formalisms, and (4) the handlers allowed in the specification. The MOP framework provides a base for its instances to build upon. One of these building blocks is the generic MOP syntax which each MOP instance specializes in a form that conforms to the problem domain by defining syntactic categories (non-terminals). Figure 2.2 shows the shared MOP syntax [7] which uses Extended Backus-Naur Form (EBNF) [1]. According to the grammar, non-terminals are surrounded by "⟨" and "⟩". Braces ("{" and "}") indicate the portion enclosed may appear zero or more times. Brackets ("[" and "]") indicate the portion enclosed is optional.

Here, we will explain the common syntax constructs in MOP:

⟨**Specification**⟩ It describes the generic MOP specification syntax, which is the base for its MOP instance- and logic-specific counterpart.

⟨**Event**⟩ Its declaration serves two purposes. First, it makes it possible to refer to in the ⟨Property⟩, and second, it may have arbitrary code (⟨Instance Action⟩) declared along that is run whenever the event is observed. The associated code helps modify the program or the monitor state.

⟨**Property**⟩ MOP specifications may contain zero or more properties. As the syntax of ⟨Property⟩ suggests, it consists of a named formalism (⟨Logic Name⟩) and a property specification using the named formalism (⟨Logic Syntax⟩). If the specification does not have a property declared, then it is called *raw*. If the user does not find the available logic plugins

Shared syntax

⟨*Specification*⟩ ::= {⟨*Instance Modifier*⟩} ⟨*Id*⟩ ⟨*Instance Parameters*⟩ "{"
{⟨*Instance Declaration*⟩}
{⟨*Event*⟩}
{⟨*Property*⟩
{⟨*Property Handler*⟩}}
"}"
}

⟨*Event*⟩ ::= "*event*" ⟨*Id*⟩ ⟨*Instance Event Definition*⟩ "{" ⟨*Instance Action*⟩ "}"

⟨*Property*⟩ ::= ⟨*Logic Name*⟩ " : " ⟨*Logic Syntax*⟩

⟨*Property Handler*⟩ ::= "@" ⟨*Logic State*⟩ ⟨*Instance Handler*⟩

Instance-specific syntax

⟨*Instance Modifier*⟩ ::= ⟨*Id*⟩

⟨*Instance Parameters*⟩ ::= ⟨*JavaMOP Parameters*⟩ | ⟨*BusMOP Parameters*⟩ | ⟨*ROSMOP Parameters*⟩ | ...

⟨*Instance Declaration*⟩ ::= ⟨*JavaMOP Declaration*⟩ | ⟨*BusMOP Declaration*⟩ | ⟨*ROSMOP Parameters*⟩ | ...

⟨*Instance Event Definition*⟩ ::= ⟨*JavaMOP Event Definition*⟩ | ⟨*BusMOP Event Definition*⟩ | ⟨*ROSMOP Parameters*⟩ | ...

⟨*Instance Action*⟩ ::= ⟨*JavaMOP Event Action*⟩ | ⟨*BusMOP Event Action*⟩ | ⟨*ROSMOP Parameters*⟩ | ...

⟨*Instance Handler*⟩ ::= ⟨*JavaMOP Event Handler*⟩ | ⟨*BusMOP Event Handler*⟩ | ⟨*ROSMOP Parameters*⟩ | ...

Logic-plugin-specific syntax

⟨*Logic Name*⟩ ::= ⟨*Id*⟩

⟨*Logic Syntax*⟩ ::= ⟨*FSM Syntax*⟩ | ⟨*ERE Syntax*⟩ | ⟨*LTL Syntax*⟩ | ⟨*PTLTL Syntax*⟩ | ⟨*CFG Syntax*⟩ | ⟨*PTCaRet Syntax*⟩ | ...

⟨*Logic State*⟩ ::= ⟨*FSM State*⟩ | ⟨*ERE State*⟩ | ⟨*LTL State*⟩ | ⟨*PTLTL State*⟩ | ⟨*CFG State*⟩ | ⟨*PTCaRet State*⟩ | ...

Figure 2.2: MOP Syntax

expressive or efficient enough, (s)he may opt for a raw specification and embed custom monitoring code inside the ⟨Instance Action⟩.

⟨**Property Handler**⟩ Handlers include arbitrary code from the instance source language to be invoked when a certain logic state or category is reached.

The following constructs may differ for each MOP instance:

⟨**Instance Modifier**⟩ These are specific to the language each MOP instance supports. Syntactically, they can be any valid identifier the language of the instance allows. They change the behavior of the monitoring code.

⟨**Instance Parameters**⟩ If present, they make a specification parametric, using the MOP instance language. However, parametricity typically depends on the language and not all MOP instances are parametric, therefore this non-terminal may be empty.

⟨**Instance Declaration**⟩ This is another portion that is specific to the language supported in the particular MOP instance. Instance declarations correspond to the declarations of monitor-local variables.

⟨**Instance Event Definition**⟩ These define the conditions under which an event is triggered. They are again specific to the MOP instance language.

⟨**Instance Action**⟩ Actions are arbitrary code associated with events and they are executed when the events they are attached to are observed. An action may modify the running program or a monitor state. The syntax of the allowed statements depend on the particular MOP instance. These statements usually differ in variables and functions they refer to compared to the ones used in handlers. This is the reason why there are separate non-terminals for actions and handlers.

⟨**Instance Handler**⟩ These are arbitrary code that are executed when a property handler is triggered.

The rest of the constructs are logic plugin-specific:

⟨**Logic Name**⟩ It is an identifier that declares the logic of the property.

⟨**Logic Syntax**⟩ It refers to the syntax of the actual property definition for each plugin.

⟨**Logic State**⟩ They refer to monitor stages or categories a handler may be written for. They are declared as constants and their definition is property-specific.

Another building block that the MOP framework provides for its instances is the logic plugins. Every logic plugin is an implementation and encapsulation of a monitor synthesis algorithm for a particular specification formalism. A set of events and a formula or pattern based on the formalism are fed to the logic plugin in order to get an output of an abstract monitor which checks a trace of events against the given formula. More information on various MOP logic plugins can be found in [16].

# Chapter 3

# Motivation and Case Study

Robots now are employed in so many areas. Examples include military robots, medical robots, and home security robots. Despite such diverse application domains, the need for their security and safety is a subject agreed upon by all researchers from all kinds of fields, and of course end users, too.

To go over a few of the concerns, let us first emphasize that even a minute mistake in the mechanics of the robot or the software that operates it, means a huge risk against lives that conduct the robot, get serviced by the robot or simply anyone within its range. Take, for example, the instance of the semi-autonomous robotic cannon that was deployed by the South African army in October 2007 during a shooting exercise [17]. Due to a probable software glitch, the computerized gun went out of control and started shooting. This overlooked error ended up taking 9 lives and injuring 11 others. Another case where precision and submissiveness have utmost importance is the field of medicine. Nowadays, robotic arms are trusted to perform vital surgical tasks on patients. Even though engineers try their best to meet the functionality and reliability requirements, the variety of purposes and the complexity it brings in terms of choice of sensors and actuation capabilities, may lead to unfortunate yet critical oversight during their composition. Our motivation is that when the possibility of errors is undeniable, there should be another level of safety which oversees the system in action and interferes with it in order to prevent dire consequences. We believe this can be achieved by monitoring the system at runtime. There is one such work focused on medical robotics which agrees that runtime monitoring is indeed a viable solution [14].

To name another concern, hacking is a serious attack that should be looked out for [18]. All the favorable features of a robot that make people want to use it in the first place, such as its strength or surveillance, could be turned against them if the robot is prone to hacking. Consider home security robots equipped with surveillance devices. If the control of these robots and/or

devices is taken over by unauthorized parties, the very assistant of yours that you trust for your protection, could easily become a threat to your privacy. In the case of military robots, hijacked sensors or controllers may cause even more disastrous situations. Therefore, it is important to supply the system with an adequate security protection mechanism.

We tested our framework's monitoring and access control capabilities on a robot called LandShark. The LandShark UGV has an onboard Linux box running ROS. Furthermore, it is equipped with various devices, such as a GPS sensor, a radar, cameras, motor and turret controllers, and a paintball gun. Each device has a driver and a corresponding ROS node which publishes sensor data and/or subscribes to topics to receive ROS messages that command the operation of the robot. An operator control unit (OCU) node listens to messages from the robot and sends it user commands.

Here, we will talk about two of the monitors we developed for LandShark.

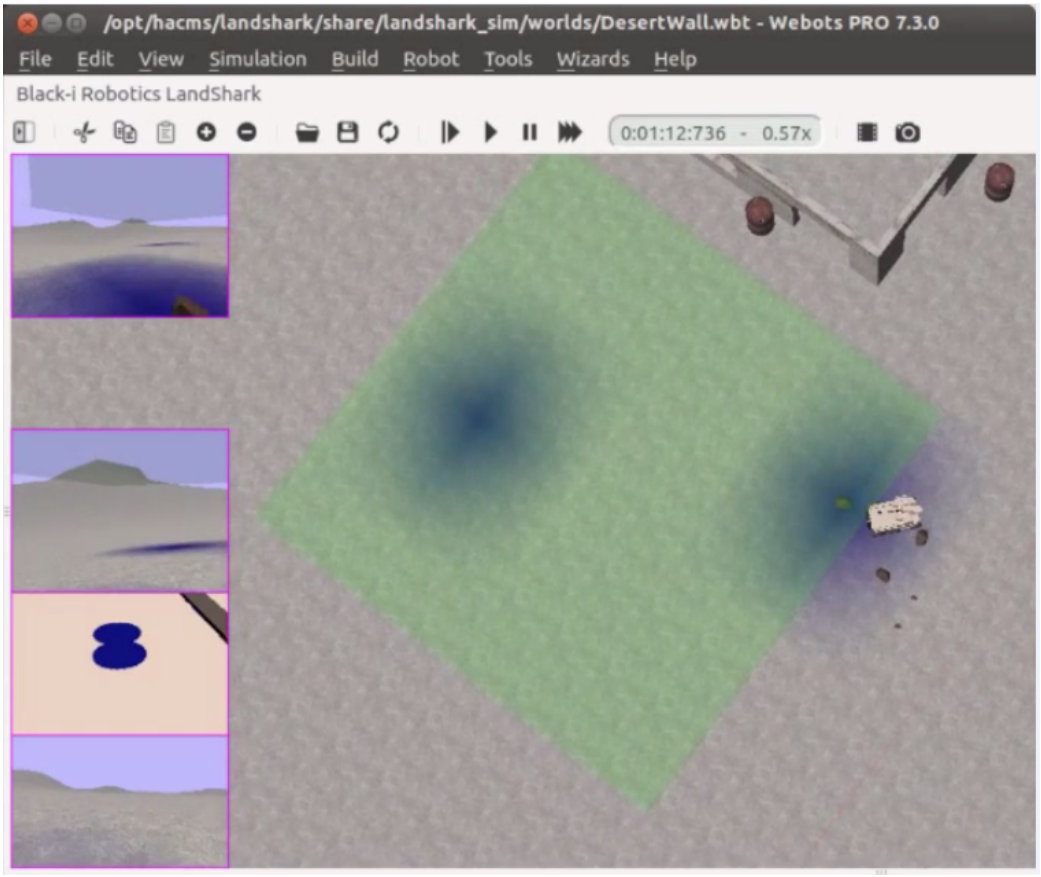


Figure 3.1: LandShark not allowed in the green zone

The figures included are screenshots taken from the Webots[1] simulator showing a replica of the robot.

Figure 3.1 shows a green area where the robot is not allowed to enter. Upon reaching the border of this unsafe zone, even though the operator commands the robot to move forward by pressing the associated button on the OCU, these messages are dropped by the monitor to prevent the command from being conveyed. For clarity, the user-specified monitor can also print out a message indicating that this move is prohibited.

Figure 3.2: LandShark shooting itself

Figure 3.2 shows the scenario where LandShark shoots itself because it does not have a mechanism that checks whether the gun is pointing at itself or not. This problem may seem trivial as one might suggest to implement an easy check as part of the gun turret driver software. However, our focus is not figuring out missing elements or criticize the lack thereof. We still think that safety-critical robots may greatly benefit from monitoring, especially when there are examples of even thoroughly tested systems sometimes failing at runtime. For this example, the monitor again drops the messages coming from the OCU that were to be received by the driver and interpreted as the trigger command, if the gun happens to be aimed at the robot.

[1]Webots is a development environment used to model, program and simulate mobile robots (`www.cyberbotics.com`).

In addition to monitors, we also tested our access control policy on LandShark. Although safety concerns can be addressed by monitors, there is still a security deficiency in ROS. As nodes can be replaced rather easily, it is possible for attackers to fake a driver node or the OCU and misdirect the robot. Our solution for avoiding malicious conduct is supplying a configuration file in which the user specifies trusted IP addresses that correspond to various devices controlling the robot. In our attempts to replace the OCU of LandShark, we successfully failed and demonstrated our access control policy works as expected.

# Chapter 4

# ROSRV

ROSRV [9] is designed to address the safety and security issues in ROS-based robot applications. Figure 4.1 shows an overview of the ROSRV communication architecture [13]. As depicted, the main difference between ROS and ROSRV architectures is the RVMaster proxy node which manages the insertion of monitors in the middle of communication channels in a transparent way. This additional layer on the original system protects both the ROSMaster from a security perspective, and the safety of the application from a functional point of view. With the inclusion of this extra layer, all node requests that were meant to be received and handled by the ROSMaster are intercepted by the RVMaster, and all messages on desired topics can be monitored. Thus, the intended safety and security policies are enforced.

An important feature of the framework is that it does not require any

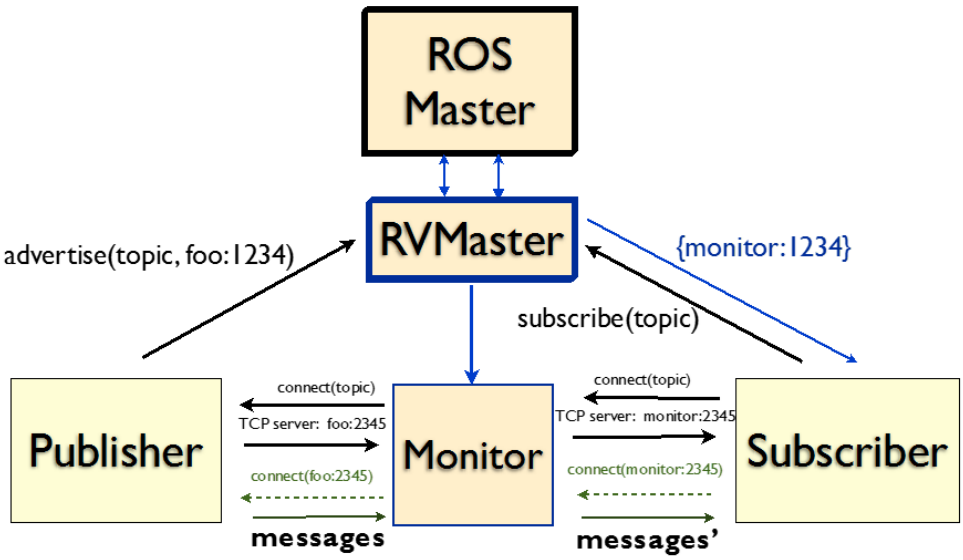


Figure 4.1: ROSRV Communication Architecture

change to the ROS source code or the application code. The only requirement for ROSRV to work with ROS is to configure the *ROS_MASTER_URI* environment variable. This variable corresponds to the host:port of the XMLRPC server the ROSMaster runs. All other nodes communicate with the ROSMaster using the default port that it binds to. By binding the RVMaster to the standard port that the ROSMaster binds to by default, all XMLRPC calls meant for the ROSMaster are directed to the RVMaster. In the meanwhile, `ROS_MASTER_URI` is configured so that the ROSMaster listens at a hidden port that is only visible to the RVMaster. This is implemented by installing a firewall that blocks access to the new ROSMaster port. With the help of this configuration, the rest of the system remains the same; nodes continue to communicate with the *"Master"* in the same way, i.e. by sending XMLRPC requests to the default port. Moreover, since the RVMaster becomes the proxy for the ROSMaster, by manipulating URIs, it can insert monitors in between ordinary publishers and subscribers without them being aware.

Figure 4.2 shows the components of ROSRV. Two of them, monitor specifications and access control policy, are inputs supplied by the user. Monitor specifications are written using an expressive formal language to define safety properties. These specifications are parsed by the ROSMOP tool and automatically converted into monitors that the RVMaster integrates into the system and orchestrates at runtime. An access control policy configuration

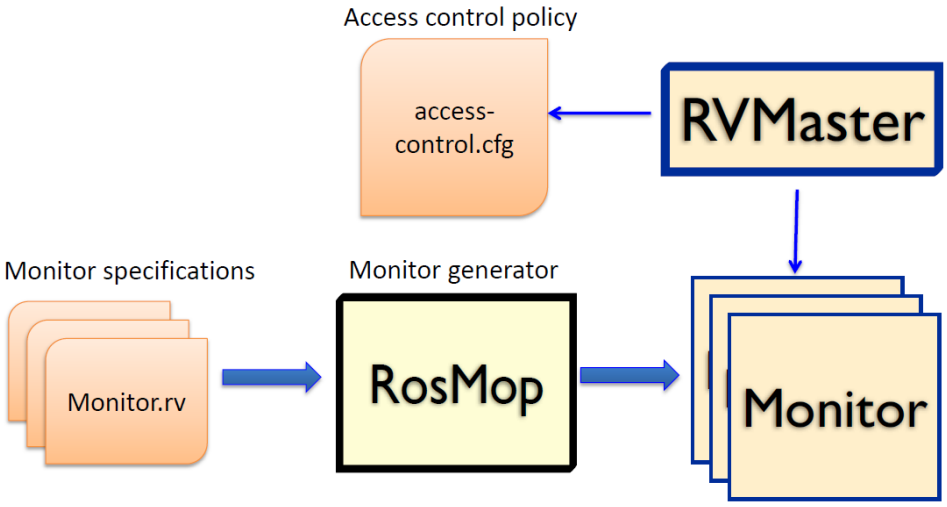


Figure 4.2: System Overview

provided by the user is enforced by the RVMaster to ensure that nodes may inquire about or alter the state of the robot as long as they are allowed to do so and their declared authenticity is not breached.

In the rest of this chapter, we will talk about the individual components of ROSRV in more detail.

## 4.1 RVMaster

The main component of ROSRV is the node RVMaster. It is the main component because, other than its most important role of being in control of enforcing security policies and monitoring safety properties, it is the reason why ROSRV can be defined as transparent. First of all, its only requirement being the reassignment of ports is what makes ROSRV's integration with ROS seamless. Secondly, its smart algorithm for managing monitors, including their activation/deactivation and placement in between other nodes, is the basis for why monitors are not detected in the system as extraordinary processes, but perceived as every other node publishing/subscribing to topics.

Despite its crucial role, the logic behind it is actually rather simple. Although it acts like the *"Master"*, in reality, it does not take over what the ROSMaster does in terms of bookkeeping. At the implementation level, the RVMaster wraps the Master API [12] that ROS client libraries call for node registration/unregistration or to retrieve system state information. Upon receiving XMLRPC calls from nodes, it checks these requests against the user-provided access control policy configuration to see whether that particular node should be granted what it asks for. If it is the case that it does not violate the specified security policy, the RVMaster makes a call to the ROSMaster itself with the same parameters. This way, the security of the robot is protected by not letting the ROSMaster execute each and every (possibly malicious) command coming from arbitrary nodes. Furthermore, by only wrapping the API, the RVMaster remains comparatively lightweight as it does not implement all the name service features the ROSMaster is responsible for. The specifics of access control policy configuration in terms of node identification and types of requests is discussed in Chapter 4.3.

System monitoring is a bit trickier than administering XMLRPC calls for access control. The role of the RVMaster here is to keep track of all the

communication requests by nodes so that monitors can be injected when necessary to those peer-to-peer communication channels of interest as men-in-the-middle. Once ROSMOP automatically generates the ROS-appropriate monitoring code (further discussed in Chapter 4.2.1) out of user-defined specifications, it is the RVMaster's job to manage them, from their initializations to their activation statuses at runtime. Although monitors act like ordinary nodes, their creation is not handled by ROS client libraries. The RVMaster implements the XMLRPC API in order to have full control over monitors. In the usual case, when nodes are written using the ROS client libraries, their interaction with the ROSMaster and the initiation of topic transports with other nodes are handled by those libraries. This means in most cases, users are not expected to code using the XMLRPC API for their nodes to connect to others. The ROS client libraries conveniently hide the details of this type of communication from users, and handle it for them. Monitors, on the other hand, are created directly from inside the RVMaster, and their communication with the rest of the system is managed through explicit XMLRPC calls. This makes it easier for the RVMaster to deal with callbacks when it needs to insert monitors and yet does not want to let other nodes know about them.

The following is how monitoring works in the system (for a better understanding, please read about the ROS inter-node communication in Chapter 2.1): All monitors are compiled with the RVMaster before it is started. When the RVMaster starts -the ROSMaster starts automatically at this point- it initializes all available monitor nodes in the system and registers them as subscribers to topics. However, this does not mean that they are active; the user is given the option to activate or deactivate any monitors at any given time during execution. This design prevents the overhead caused by monitoring from affecting the performance when it is not desired. We will assume that we have one monitor on one topic and it is activated from this point forward. If at this point, there are no other nodes interested in the topic, it simply does not do anything; as an ordinary subscriber would do, it waits for a callback that will inform it of newly joined publishers. When a publisher registers on the same topic, the monitor receives its URI in order to initiate a topic transport negotiation. After the connection is established between the two, all messages from the publisher are received by the monitor.

If the first registered node after the system start and monitor registration, is instead a subscriber, the RVMaster tells the ROSMaster to register

the subscriber, but it makes a note that when a publisher becomes available on that topic, the callback the subscriber receives about this situation will include the monitor's URI instead of the actual publisher's. This way, the monitor will receive all messages coming from the publisher, and after observation and potential modification, it will send them to the subscriber pretending like it is the original publisher. Note that, monitors do not create and send messages themselves naturally. They only relay messages coming from publishers. That is why when the first node available is a subscriber, a connection between the subscriber and the monitor is not established right away, and rather postponed until after a publisher registers.

Since the RVMaster has all the URI information it needs and it intercepts all XMLRPC requests received from nodes, when nodes want to query the system state about available publishers/subscribers, the RVMaster can successfully hide the presence of monitors. In addition, monitor activation/deactivation at runtime is again simply handled by callbacks. Alerting publishers/subscribers that the connection information of their correspondents has changed easily does the job when a monitor becomes active and needs to intercept messages, for example. This situation being common in ROS -as nodes may be replaced when a second node with the same name is introduced, this has to happen anyway- is the beauty of it.

The RVMaster also supports additional XMLRPC calls for users to be able to query the state of monitors, especially for debugging purposes. The two options it provides are (1) to list the available monitors and their activation statuses, and (2) to give more comprehensive information about the runtime verification state. This includes, the ports the RVMaster and the ROSMaster bind to, and a list of all the monitors with their node names, the topics they monitor, the ports of their XMLRPC server, and lastly their (usually) TCP ports that they use to connect to other nodes.

On a note about our earlier design approach of monitors, they were first considered as merely interceptors in between publishers and subscribers on topics which were only concerned about messages sent and received on that particular topic. However, later on we realized that this approach falls short when users want to publish a message from inside the monitor on another topic based on the information carried by the received message. For example, one may want to monitor messages containing the information of a robot's position and when the robot moves out of a restricted area, an alarm message

on a different topic needs to be sent. For this purpose, we had to change both our design and implementation. Our current method handles this issue by keeping a pointer map of topics shared among monitors, to be able to publish to any of them at any time as needed. This requirement also enabled us to implement better monitors which act more like ordinary ROS nodes.

Next, we will focus on the user side of monitoring, as we will explain how specifications are converted into monitors that the RVMaster can handle.

## 4.2 Monitoring Safety Properties

The RVMaster manages monitors at runtime in order to protect the application functionally, but it does not generate them itself. ROSMOP is the tool that takes monitor specifications and generates C++ code that the RVMaster can work with. In other words, they can be initialized, activated and deactivated by the RVMaster, but if the user specifies a certain action to be taken or requires a computation based on contents of messages intercepted, this is incorporated into the monitor by the help of ROSMOP.

In this section, we will first take a look at ROSMOP in detail and explain how user-defined specifications in MOP syntax are converted into C++ monitoring code and correspond to callbacks in which messages are received and sent. Then, one of the monitor specifications will be explained in detail to demonstrate how easy it is to construct a monitor from user's perspective.

### 4.2.1 ROSMOP

ROSMOP [8] is an MOP instance specifically designed to integrate monitors into the ROS framework. Its current implementation is devised to work within ROSRV, as its generated monitoring library is particular in its design to be used by the RVMaster. ROSMOP can take multiple specifications as input at a time, and carefully handles their complexity to make sure there is only one callback generated per topic in the end. This is because ROS allows only one callback registration per topic. However, with ROSMOP keeping the metadata when merging all monitoring code under one callback, the RVMaster can still support multiple monitors defined on a single topic. In some cases, the user is warned before the library generation is attempted if there are any possible complications foreseen due to contradictory defini-

⟨*ROSMOP Specification*⟩ ::= ⟨*Id*⟩ "(" ⟨*C++ Parameters*⟩ ")" "{"
[⟨*C++ Declarations*⟩]
⟨*Event*⟩$^+$
{⟨*Property*⟩
{⟨*Property Handler*⟩}}
"}"

⟨*Event*⟩ ::= "*event*" ⟨*Id*⟩ "(" ⟨*C++ Parameters*⟩ ")"
⟨*ROS-related Parameters*⟩
"{" ⟨*Extended C++ Statements*⟩ "}"

⟨*ROS-related Parameters*⟩ ::= `topic message "'{"parameter-message access pattern"'}"`

⟨*Property*⟩ ::= ⟨*Logic Name*⟩ " : " ⟨*Logic Syntax*⟩

⟨*Property Handler*⟩ ::= "@" ⟨*Logic State*⟩ "{" ⟨*C++ Statements*⟩ "}"

Figure 4.3: ROSMOP Syntax

tions, such as the same global variable declaration in multiple specifications. After successful generation of the monitoring library, it is placed correctly in ROSRV so the RVMaster works smoothly at runtime. The RVMaster is factored in such a way that the monitoring code is separate from the other components. The monitoring library and the RVMaster communicate through a predefined API, so that even when new monitors are generated by ROSMOP, the RVMaster codebase still remains unchanged. Therefore, all in all, the only thing users need to do to monitor their ROS setup, is to supply the specifications and push the button to watch it in action.

ROSMOP specification files end with the extension ".rv". In one file, there may be multiple specifications defined. Figure 4.3 shows the instantiated ROSMOP syntax. According to this syntax, each named ROSMOP specification consists of optional global variable declarations, one or more events, and zero or more properties.

Global variables declared may facilitate interaction between events. For example, one might need to monitor one topic in order to get certain information about the robot, and according to that information, (s)he may need to interact with messages on another intercepted topic. This can be done by allocating a global variable. Also, in the case of multiple specifications parsed at once, these global variables get merged into one place. So, they can actually be helpful in getting monitors to collaborate as well, in addition to only across events in a single monitor.

As every other registration XMLRPC call is handled, monitors, too, need

to provide topics and their corresponding message types. This is incorporated into the event signature in ROSMOP. Moreover, ROSMOP supports parametric events in order to access the specified message's fields. To bind to those fields, parameters need to be matched in the provided pattern to the appropriate fields. This way, the user may observe and optionally modify the message content through declared references.

Extended C++ statements constitute the event action. Event action is where users may define what monitors will do after observing message content. If the intended action is to log the messages, for example, they can do so by writing C++ code in the event action, and the monitor will do just that with the message before it relays it to subscribers. Therefore, event actions are where users may dictate monitors to do extra computations or change the monitor status. The reason these C++ statements are referred to as *extended*, is because we incorporated two additional keywords into the language, which are PUBLISH and MESSAGE. PUBLISH keyword helps users when they want to publish to another topic from inside the event action of the current topic. By using this keyword, users can make monitors publish to any other available topics which may or may not be intercepted by that monitor through its events. In some cases, we saw that all the fields of a message need to be accessed and stored in a global variable for its use in another event. Instead of doing so by writing out all the necessary parameters and matching them in the message pattern section, MESSAGE keyword does that automatically for you.

ROSMOP currently supports two logic plugins: FSM [4] and CFG [3]. The FSM plugin encapsulates the monitoring algorithm where the current state of a deterministic finite state machine is represented by an integer. With the observation of each event, the value of the current state and the received event together determine the next state of the finite state machine. The CFG plugin allows users to create monitors out of context free grammar descriptions. It uses a modified version of the standard table driven GLR parsing algorithm. The synthesized monitors are based on push-down automata. Logic plugins support either predefined states or uses of aliases to describe states. One can match these states by using "@ $\langle$*Logic State*$\rangle$" to attach arbitrary C++ statements to define actions in property handlers which are executed upon validation or violation of the property.

As mentioned, when there are multiple specifications as input, and multi-

ple events on one topic, those event actions get merged to output only one callback per topic. ROSMOP names monitors after their specification IDs. So users are encouraged to think that each single specification corresponds to one monitor. Even though, the event actions are merged to output a single callback to comply with ROS standards, the information about which event belongs to which specification (hence, which monitor) is kept during the transformation. Therefore, at runtime, the RVMaster can still manage the execution of designated event actions even when a certain monitor is not active but the other is, which both have an action declared on the same topic.

Next, an example monitor specification will be explained in order to make more sense of the ROSMOP language syntax in terms of applicability.

### 4.2.2 Monitor Specifications

After going over the language syntax, here we will analyze the safe triggering monitor specification that was mentioned in Chapter 3.

```
safeTrigger() {
     bool isSafeTrigger = false;
     event checkPosition(string N, double P)
            /landshark/joint_states sensor_msgs/JointState
           ‘{name[1]:N, position[1]:P}’  {
     if (N=="turret_tilt"){if (P > -0.45){ //check gun position
                        isSafeTrigger = true;
                  }else{
                        isSafeTrigger = false;
     }     }    }
     event safeTrigger() /landshark_control/trigger
            landshark_msgs/PaintballTrigger ‘{}’  {
            if(!isSafeTrigger) return; //drop trigger message
     }
}
```

Figure 4.4: Safe Trigger Specification

As explained in the previous section, safety properties imposed as monitors at runtime are defined by means of specifications basically consisting of events, actions, and properties (omitted in raw specifications). Figure 4.4 shows an example of a specification to illustrate the idea. This specification is *raw*; it does not have an explicit property declared. Instead, user-defined

custom monitoring code is embedded into event actions which can be any C++ code.

The safety condition we want to monitor here requires that the robot can only fire in certain safe poses. If the gun happens to be directed at the robot, it results in the message being dropped by the monitor in order to prevent triggering. There are two events, `checkPosition` and `safeTrigger`, which listen to messages on two different topics. On each topic, there can only be a certain type of message sent and received, which is also provided in the event signature.

In this specification, the `checkPosition` event checks whether the gun is at a safe position to trigger. This means that it should not be pointing at the robot. This condition corresponds to its angle being greater than 45 degrees. For this purpose, the `safeTrigger` monitor listens to the topic `/landshark/joint_states` with the message type `sensor_msgs/JointState`. Message fields can be accessed by providing necessary parameters as done here; there are two arrays in `sensor_msgs/JointState`, `name` and `position`, which are bound to variables `N` and `P`, respectively. These parameters are used in the action code of the event to check the validity of the safety condition by observing the message content.

By the help of event handlers and parameters, monitors can not only observe the message content, but also decide to either modify the message value or drop it altogether, or trigger some other action for that matter. For example, in `checkPosition`, the global variable `isSafeTrigger` is set to true if and only if the gun is at an angle larger than 45 degrees. At the same time, this variable is checked in the `safeTrigger` event to determine whether the gun is allowed to trigger or not by either relaying the message as is, or not sending it at all.

## 4.3 Enforcing Security Policies

Although ROSRV has always been considered as a monitoring framework from day one, due to its requirement of overseeing all communication between nodes and the ROSMaster, and as a result wrapping the Master API, it turned out to be an ideal place to incorporate access control as well. Access control is meant to rightfully qualify nodes in the system to be allowed to

carry certain actions. All actions a node may attempt are bounded by the supported XMLRPC API, and ROS, by default, grants all nodes full access. However, this situation leaves ROS vulnerable in the face of an attack, as it makes it very easy for attackers to seize the control of a robot. With our solution, we give the users the privilege to restrict arbitrary nodes' capabilities to protect their robot from being exposed.

The access control policies are provided as input in a configuration file to the RVMaster for their enforcement at runtime. Upon receiving node requests, the RVMaster checks them for compliance to security configurations, and if it finds them appropriate, it sends them to the ROSMaster for issuing. For example, a node registering for publishing to a certain topic may or may not be allowed in the provided configuration, and if the RVMaster decides that it is not by checking the node identity and the topic name in the request, it rejects the registration by not passing the request on to the ROSMaster.

Nodes are identified by their IP addresses in the configuration file, instead of their node names, to prevent attackers from faking a node. From the RVMaster's point of view, the identity check of a node is done by extracting the IP address of the node from its XMLRPC request. However, since normally the xmlrpcpp [11] library provided in ROS does not expose the IP address information in RPC invocations, we extended this library and included it as part of ROSRV. The use of IP addresses implies access granularity at host level. In order to enhance user experience, we support definitions of IP aliases and groups so that users do not need to repeatedly deal with IP addresses in the access control configuration.

Currently, there are four main policy categories available for use. These are `[Nodes]`, `[Subscribers]`, `[Publishers]`, and `[Commands]`. Under each category, the access policies are written as a key followed by an assignment symbol and a list of values. The following shows what each key and value list pair corresponds to:

`[Nodes]:` *key* = node name, *value* = machine identity allowed to create the specified nodes

`[Subscribers]:` *key* = topic name, *value* = node identity allowed to subscribe to the topic

`[Publishers]:` *key* = topic name, *value* = node identity allowed to publish

to the topic

`[Commands]`: *key* = command name, *value* = node identity allowed to perform the command

Next, we will look at an example access control configuration written for our case study robot LandShark.

### 4.3.1 Access Control Policy Configuration

```
[Groups]
localhost = 127.0.0.1
certikos = ip1 ip2 ip3 ip4
ocu = ip5 ip6 ip7 ip8
[Nodes]
default=localhost
/landshark_radar=certikos

[Publishers]
default=localhost certikos
/landshark_control/trigger= ocu
[Subscribers]
default = localhost certikos
/landshark/gps = ocu

[Commands]
# Commands: full access
getSystemState = localhost certikos ocu
# Commands: limited access
lookupNode = localhost certikos
# Commands: local access only
shutdown = localhost
```

Figure 4.5: Access Control Policy Configuration

Figure 4.5 shows a snippet of the LandShark access control policy configuration. The `[Group]` section defines three groups over 9 IP addresses and gives them aliases. This indicates, for example, that accesses granted to the OCU are only valid as long as they come from these specified IP addresses.

In the `[Nodes]` section, `default = localhost` means that by default the machine `localhost` is allowed to create a node with any name. Names can also be precise, as in `/landshark_radar = certikos`, which means that the alias `certikos` is allowed to create a node with the name `/landshark_radar`.

In the `[Publishers]` section, only nodes running on machine `ocu` can publish to topic `/landshark_control/trigger`. There are two ways an attacker is blocked with the combination of sections supported in the configuration. For example, even when an attacker can query the system state to get node names in order to impersonate them by replacing them, in this case knowing the name of a node would not help the attacker as topic publishers and subscribers are explicitly specified by their IP addresses. So even if the attacker finds out about the name of the node publishing to topic `/landshark_control/trigger` and creates a node with the same name, as long as it does not have one of the IP addresses the alias `ocu` covers, it cannot register to this topic as a publisher. The second way it may be blocked is that under `[Nodes]` section, `ocu` alias may protect the node names it allocates for its purposes. Therefore, even if the attacker can figure out the

names of nodes publishing to topics it wants to pose as, it would not be able to create nodes with those names.

In `[Commands]`, `getSystemState = localhost certikos ocu` means that nodes running on machines `localhost`, `certikos`, or `ocu` are allowed to send `getSystemState` requests to the ROSMaster, and `shutdown = localhost` means that only nodes on `localhost` are allowed to shutdown other nodes.

# Chapter 5

# Evaluation

We have evaluated our framework according to the following research questions.

**RQ1 - Is the specification language expressive?**

Throughout the development of this project, we have had the chance to collaborate with other teams who were working with ROS on the same robot, LandShark. Since we were looking for ideas to test our framework's capabilities, in particular, our specification language since it is the first step in defining monitors, it was very convenient for us to work with researchers who were not involved in the design process of ROSRV. It was especially a good match when we found out that one of the teams was also dealing with safety concerns, such as obstacle avoidance. We took this suggestion as an opportunity to test what ROSRV offered and what needed improvement.

We took four scenarios into consideration for evaluation that one might ask from a monitoring framework. The first application we tried was logging, as ROSRV monitors already intercept all messages on a given topic and observe their contents. This is trivial to express with the specification language; one can simply access message fields of interest by specifying parameters (as explained in Chapter 4.2.2), and if desired, log them by writing C++ code and including it in the event action.

The second scenario we were interested in was safe triggering, because this is also a valuable monitor for real-life applications, such as military robots. By using a simple concept like global variables, it was easily possible to specify this safety property which needed two different topics to be monitored.

The third condition was to keep LandShark in a safe zone and not let it move out of the restricted area. This was also quite possible with our specification language. We needed to monitor the messages which carried odometry, GPS

and velocity information. By accessing the message fields, we were able to check whether the robot's GPS data indicated that it was inside of the safe zone or not. If the GPS and odometry monitors detected that the robot passed beyond the border of the designated area, by setting a global flag, the velocity monitor was alerted to drop the message to prevent the robot from moving forward. This was a convenient scheme to demonstrate monitors' capability of taking a collaborative action based on information coming from more than one sensor.

As mentioned, the last one we tested was obstacle avoidance. This one was the most complex among others. It required checking certain conditions periodically, and with precise timing. It was about recalculating a radius of distance based on how fast the robot was moving at a certain time and stopping the robot instantly at that distance if an obstacle was perceived. It was actually while we were defining this property that we decided to add the two keywords PUBLISH and MESSAGE to ROSMOP. MESSAGE keyword was useful in copying all the values of a message's fields to a global variable with the same type, without explicitly accessing all the fields. PUBLISH keyword was used in this case to send an alarm message on a different topic whenever an obstacle was encountered. All in all, on top of proving that our specification language was capable of expressing intricate properties, this also showed that with challenging examples and user requests, the design can be further improved.

**RQ2 - What is the overall performance?**

To assess the overall performance of ROSRV, we conducted several simple performance tests. For this purpose, we implemented two nodes, one a simple publisher and the other a simple subscriber [21], in order to test the overhead of introducing monitors to the system. The only purpose of this insignificant setup was to measure the average number of messages delivered to the subscriber with and without the presence of monitors to see how the system performs in both cases in a certain amount of time. Therefore, we did not include any time-consuming computations as part of the execution cycles of the nodes.

In our first experiment, we ran the two nodes which connect to each other

on a single topic for 10 seconds under 3 conditions. The first condition was to run them using the ROSMaster alone by initiating it with the ROS-provided command `roscore`. In this case, the number of messages sent by the publisher was 97, and the number of messages received by the subscriber was 93. The second condition was to run the same nodes with ROSRV by calling `rvcore`, but without activating the monitor listening to the topic. This time, the number of messages sent by the publisher was 95, and the number of messages received by the subscriber was 90. The third condition was to activate the monitor, and the numbers we got for messages sent and received were 94 and 89, respectively. In our experiments, we have seen that the first few messages originated by the publisher were not received by the subscriber at all cases; whether the system was run with or without ROSRV did not matter.

For our second experiment, we used the exact same setup, but this time we ran the nodes for 10 minutes. The numbers we got out of this experiment were the following:

**First condition (roscore)** Messages sent: 5996, Messages received: 5992

**Second condition (rvcore -no monitor)** Messages sent: 5995, Messages received: 5990

**Third condition (rvcore -monitor)** Messages sent: 5992, Messages received: 5987

As can be seen, the length of the execution time does not have an impact on the overhead the monitors impose, as the results are very similar to when the execution time was only 10 seconds.

For our third experiment, we modified these two nodes to be both publishers and subscribers. In this setup, `node1` was publishing to `topic1` and subscribing to `topic2`, and `node2` was publishing to `topic2` and subscribing to `topic1`. This way, individual nodes had extra instructions to execute. Furthermore, we had the chance to measure the impact of running more than one monitor. The conditions we ran the experiment under were the same and here are the results we obtained:

*Execution time: 10 seconds*

**First condition** `topic1`: Messages sent: 97, Messages received: 92;

`topic2`: Messages sent: 96, Messages received: 94

**Second condition** `topic1`: Messages sent: 96, Messages received: 91;
`topic2`: Messages sent: 95, Messages received: 91

**Third condition** `topic1`: Messages sent: 95, Messages received: 90;
`topic2`: Messages sent: 94, Messages received: 89

*Execution time: 10 minutes*

**First condition** `topic1`: Messages sent: 5996, Messages received: 5992;
`topic2`: Messages sent: 5996, Messages received: 5991

**Second condition** `topic1`: Messages sent: 5996, Messages received: 5990;
`topic2`: Messages sent: 5995, Messages received: 5991

**Third condition** `topic1`: Messages sent: 5993, Messages received: 5989;
`topic2`: Messages sent: 5994, Messages received: 5990

In the last experiment we conducted, to see what adding more overhead does to the performance, we used the PUBLISH keyword inside the event action to force the monitor to publish to a different topic. What adding PUBLISH to the monitor does is that instead of only sending intercepted messages to the subscriber, the monitor is now responsible for sending an extra message for each one it intercepts. We ran the simple publisher-subscriber setup for this experiment on only a single topic for 1 minute under again the same conditions. The following are the numbers we collected:

**First condition** Messages sent: 594, Messages received: 590

**Second condition** Messages sent: 596, Messages received: 590

**Third condition** Messages sent: 595, Messages received: 590

These results show that, even when monitors are busy with extra computations that users desire them to deal with, the monitoring overhead is still negligible.

Aside from these simple cases which may not be too convincing by themselves, we also tested our framework on the actual LandShark robot. The complexity of the robot is undoubtedly beyond comparison to these simple setups. At runtime, it creates more than 10 nodes corresponding to devices

and sensors on the robot, and communicates on at least 20 topics for its operation. During our experiment, we activated two monitors covering 5 of the topics, and the overhead we measured was not more than a few milliseconds. This demonstrates that ROSRV's applicability is definitely substantial and the overhead it incurs is tolerable.

On a related note about performance, the current implementation of ROSRV is centralized. This means that all the monitor nodes live in the same multi-threaded process. Although in our evaluation with simple tests and the overall performance we achieved with the LandShark robot, we have found the message delay caused by monitoring acceptable, the centralized design may face scalability issues when a more complex robot is in question. In the future, a decentralized mechanism, such as using a multimaster [22], may be considered to improve scalability. This approach would also enhance the fault tolerance of the system, as the current centralized master design is a single point of failure.

**RQ3 - Is the access control effective in restricting nodes?**

To assess the capability of our access control implementation, we arranged a multi-machine setup, where only one of the machines ran the RVMaster and the other connected to that one by assigning the correct host:port value to the environment variable `ROS_MASTER_URI` after establishing ssh connections between the two machines. With this setup, and a given access control policy that restricts publishing to `topic1` to only nodes created on the same machine with the RVMaster, we tried to register a publisher from the second machine. Upon receiving the XMLRPC request and checking it against the policy configuration, the RVMaster successfully printed out a message indicating that this request cannot be carried out. Moreover, we also tested this feature on LandShark, by trying to kill arbitrary nodes running on the robot from an external source. As shown in Figure 4.5, the execution of `shutdown` command is performed only when the request comes from one of the nodes of the robot itself. Therefore, our attempt to kill any of the robot's nodes failed as expected due to this restriction.

Although our access control is effective in allowing and disallowing nodes' communication with the ROSMaster the way the user configures, the main

limitation of our implementation is the dependence on IP addresses and network routing to ensure security. Our design currently naively trusts IP addresses. Therefore, it is not protected against possible attackers who can impersonate trustful sources by spoofing IP addresses or run processes on the same (virtual) machines as trusted nodes. One way to defend against such attacks would be to run the RVMaster on a separate (virtual) machine than where (possibly distrustful) nodes are run. Another, and maybe a complementary way, would be to configure the machines to communicate with each other using additional security schemes, such as encrypted tunnels. We have looked into IPsec [15] for this purpose, and confirmed that ROSRV can work with it and benefit from it. However, we have not yet developed an automation of such a configuration to provide this level of security in a more user-friendly way.

# Chapter 6

# Conclusion

The undeniable and ubiquitous presence of robots will always mean that there might be occasional disasters waiting around the corner. To minimize the damage and possible disasters themselves, we need to make sure that robots operate safely and securely. This thesis presents ROSRV, a runtime verification framework for the Robot Operating System (ROS). Favorable features of ROS make it one of the most popular robot software development frameworks, however, it does not offer a safety and security mechanism for robotic applications to depend on for a more reliable operation. ROSRV aims to address the vulnerabilities of ROS by monitoring safety properties and enforcing security policies, with a seamless integration in doing so.

As part of ROSRV, we developed ROSMOP for automatically generating monitors out of user-defined formal specifications, and a proxy node, called RVMaster, that injects them into the system transparently at runtime. The RVMaster also supervises communication of nodes with the ROSMaster through access control, in order to avoid possible malicious activity misdirecting the robot.

Our evaluations show that the formal specification language developed for users to define safety properties is expressive, the overhead introduced by monitoring is acceptable, and the implemented access control prevents the operation of the robot from being intruded.

***Future Work*** Currently the runtime verified system is not formally verified. First of all, this would require a formal model of ROS itself. Then the next step would be to guarantee that ROSRV indeed complies with this model. Internally, this requires the proof that generated monitors and glue code satisfy the desired system properties at runtime, and that the RVMaster invokes the monitors at correct times. Additionally, tools should be developed to prove that the automatically generated monitors do actually monitor the safety properties defined in specifications.

# References


[1] ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF, 1996.

[2] BusMOP — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/Special:BusMOP, 2008. [Online; accessed 9-November-2014].

[3] CFG Plugin4 Input Syntax — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/CFG_Plugin4_Input_Syntax, 2014. [Online; accessed 7-December-2014].

[4] FSM Plugin4 Input Syntax — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/FSM_Plugin4_Input_Syntax, 2014. [Online; accessed 7-December-2014].

[5] JavaMOP4 — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/JavaMOP4, 2014. [Online; accessed 9-November-2014].

[6] Monitoring-Oriented Programming — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/Monitoring-Oriented_Programming, 2014. [Online; accessed 9-November-2014].

[7] MOP4 Syntax — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/MOP4_Syntax, 2014. [Online; accessed 4-December-2014].

[8] ROSMOP — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/ROSMOP, 2014. [Online; accessed 21-November-2014].

[9] ROSRV — Formal Systems Laboratory. http://fsl.cs.illinois.edu/index.php/ROSRV, 2014. [Online; accessed 21-November-2014].

[10] Ben Axelrod. roscpp/Overview/Initialization and Shutdown — ROS.org. http://wiki.ros.org/roscpp/Overview/Initialization%20and%20Shutdown, 2014. [Online; accessed 9-November-2014].

[11] Ken Conley. xmlrpcpp — ROS.org. http://wiki.ros.org/xmlrpcpp, 2011. [Online; accessed 6-December-2014].

[12] Alexander Gutenkunst. ROS/Master API — ROS.org. `http://wiki.ros.org/ROS/Master_API`, 2014. [Online; accessed 5-December-2014].

[13] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Roşu. ROSRV: Runtime Verification for Robots. In *Proceedings of the 14th International Conference on Runtime Verification*, volume 8734 of *LNCS*, pages 247–254. Springer International Publishing, September 2014.

[14] Min Yang Jung and Peter Kazanzides. Run-time Safety Framework for Component-based Medical Robots. In *4th Workshop on Medical Cyber-Physical Systems*, April 2013.

[15] Stephen Kent and Karen Seo. Security Architecture for the Internet Protocol. RFC 4301, RFC Editor, December 2005.

[16] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14(3):249–289, 2012.

[17] Noah Shachtman. Robot Cannon Kills 9, Wounds 14 — Wired.com, October 2007. [Online; accessed 25-November-2014].

[18] Patrick Lin. The Big Robot Questions — Slate.com, February 2012. [Online; accessed 16-November-2014].

[19] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[20] Aaron Martinez Romero. ROS/Concepts — ROS.org. `http://wiki.ros.org/ROS/Concepts`, 2014. [Online; accessed 2-December-2014].

[21] Isaac Saito. ROS/Tutorials/WritingPublisherSubscriber(C++) — ROS.org. `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29`, 2013. [Online; accessed 6-December-2014].

[22] Daniel Stonier. ROS Multimaster — ROS.org. `http://wiki.ros.org/rocon_multimaster`, 2014. [Online; accessed 4-December-2014].

[23] Dirk Thomas. ROS/Introduction — ROS.org. `http://wiki.ros.org/ROS/Introduction`, 2014. [Online; accessed 9-November-2014].

[24] Victor Mayoral Vilches. ROS/Technical Overview — ROS.org. `http://wiki.ros.org/ROS/Technical%20Overview`, 2014. [Online; accessed 6-November-2014].