Spring 2019

A Framework for Test & Evaluation of Autonomous Systems Along the Virtuality-Reality Spectrum

Nathan D. Gonda
Old Dominion University, infester22@hotmail.com

A FRAMEWORK FOR TEST & EVALUATION OF AUTONOMOUS SYSTEMS ALONG THE VIRTUALITY-REALITY SPECTRUM

by

Nathan D. Gonda
B.S. May 2017, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

MODELING AND SIMULATION

OLD DOMINION UNIVERSITY
April 2019

Approved by:

James F. Leathrum Jr. (Director)

Yuzhong Shen (Member)

Yiannis E. Papelis (Member)

ABSTRACT

A FRAMEWORK FOR TEST & EVALUATION OF AUTONOMOUS SYSTEMS ALONG THE VIRTUALITY-REALITY SPECTRUM

Nathan D. Gonda
Old Dominion University, 2019
Director: Dr. James F. Leathrum Jr.

Test & Evaluation of autonomous vehicles presents a challenge as the vehicles may have emergent behavior and it is frequently difficult to ascertain the reason for software decisions. Current Test & Evaluation approaches for autonomous systems place the vehicles in various operating scenarios to observe their behavior. However, this introduces dependencies between design and development lifecycle of the autonomous software and physical vehicle hardware. Simulation-based testing can alleviate the necessity to have physical hardware; however, it can be costly when transitioning the autonomous software to and from a simulation testing environment. The objective of this thesis is to develop a reusable framework for testing autonomous software such that testing can be conducted at various levels of mixed reality provided the framework components are sufficient to support data required by the autonomous software. The paper describes the design of the software framework and explores its application through use cases.

This thesis is dedicated to my family, friends, and colleagues whom I love and respect.

ACKNOWLEDGMENTS

There are many people who have contributed to the successful completion of this dissertation. I want to thank Dr. James Leathrum for serving as my major advisor. His patience and guidance on my research deserve special recognition. I also want to extend many thanks to my committee members, Dr. Yuzhong Shen and Dr. Yiannis Papelis, for taking time to contribute their knowledge and edit this manuscript. My fellow lab members also deserve special recognition for their long hours and hard work in helping to conduct the research and development presented in this thesis. In particular, undergraduate Thomas Laverghetta contributed immensely in developing the test examples presented in this thesis. This experience was a very rewarding one, and I believe that the project greatly benefited from the group effort.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The purpose of this research is to design and create a Test & Evaluation environment for testing autonomous vehicles throughout the design and development of the autonomous software and physical hardware. The process of testing the autonomous software can be extremely difficult; therefore, it is desirable to test early in the design and throughout the development lifecycle. A framework is created in which the autonomous software can be developed such that testing can be conducted at various levels of mixed reality provided the components of the framework are sufficient to support data required by the autonomous software[1].

Autonomous systems represent an increasingly diverse and complex research area in engineering and industry. Goldman Sachs Research predicted a $100 billion dollar market just for autonomous drones from 2016-2020 [1]. The top commercial/civil sectors include construction, agriculture, finance, and public safety (police, fire, coast guard). There are efforts in universities to establish curriculums to prepare engineers for working with autonomous systems policy and risk management [2].

Autonomous systems have been utilized in numerous applications in science and technology. Studies have been conducted to understand the key factors in adopting driverless cars into the daily life [3]. There are also examples of autonomous software in non-vehicular systems such as those used in high-frequency trading strategies in the U.S. capital market [4]. Autonomous vehicles could also be sent to places that are uninhabitable by humans or placed in situations that would traditionally place a human in danger. For example, unmanned aerial

[1] IEEE Transactions and Journals style is used in this thesis for formatting figures, tables, and references.

vehicles have a role in operational forest fire activities with high maneuverability and high capacity to perform activities such as reconnaissance [5]. Autonomous systems also help by lowering the cost of maintenance by allowing the system to care for itself. For example, there are high-level control systems that carry out energy regulation on hybrid power systems [6].

However, autonomous systems present new challenges in Test & Evaluation. Koopman and Wagner discuss the inherit difficulties in testing autonomous vehicles [7]. Similar assessments are made by Menzies and Pecheur [8] and Schumann and Visser [9]. First, there is also no human backup to address faults, malfunctions, or unexpected operating conditions. Fully autonomous vehicles must have additional complexity to address all potential contingencies. Real-world testing is not able to completely validate every operating condition the autonomous software might encounter in the field, especially ones that may not be observable in the long term [7]. Second, autonomous software often utilizes non-deterministic behavior and statistical algorithms to adapt its own behavior to cope with changing surroundings [7]. As such, the software can provide different outcomes given the same test scenario. This makes it difficult to evaluate the results of testing because there is no uniquely correct result for a given test scenario and the tests are non-repeatable [8]. Third, acceptance of the behavior of autonomous systems is a critical concern. Helle, Schmai, and Strobel find that current design and testing methods are insufficient to assure safety as they make assumptions about the autonomous system behavior in the field based on the testing environment [10]. The software system must be tested extensively to demonstrate that failure rates do not exceed an acceptable safety threshold. Such vehicle testing is time consuming and expensive, and it is often not feasible to conduct enough tests to ensure desired safety level.

Simulation-based testing can work to reduce development cost by allowing testing of imperfect systems placed in hazardous or otherwise impractical situations without physically endangering people or property [10]. In a virtual, or simulated, world, complex systems and conditions can be abstracted and manipulated relatively quickly and inexpensively to represent different scenarios. This can be especially helpful in the early stages of design prior to development of a physical prototype of the vehicle. In later stages of development, virtual reality (VR) and augmented reality (AR) can gradually increase the resolution of the stimuli until real world testing is possible.

However, these approaches only work if the autonomous software is readily compatible with the operating environment with which it is tested. Often this requires many specific test harnesses to maintain compatibility with heterogeneous test environments, thus limiting the scenarios that testers can evaluate until real-world trials can commence with the fully-integrated autonomous system [11]. If the autonomous software could seamlessly integrate virtual and physical components over its development lifecycle, VR and AR can greatly reduce the cost of development by providing an easier transition between simulation-based testing and fully integrated testing.

The objective of this thesis is to develop a reusable framework in which the autonomous software can be developed such that testing can be conducted at various levels of mixed reality provided the components of the framework are sufficient to support data required by the autonomous software. The framework allows initial testing on a simulated vehicle in a virtual environment. Then as initial hardware becomes available, testing can work on a physical vehicle in a virtual environment (VR). By gradually allowing the vehicle to perceive its environment,

the vehicle can respond to the physical environment while also responding to virtual information (AR). Finally, testing transitions to full physical testing.

The discussion is composed into four main chapters. Chapter 2 describes several key background materials that are important for understanding the rest of the paper. Chapter 3 presents system requirements to support generalized testing with VR and AR and a high-level design that highlights necessary roles, responsibilities, and behaviors. Chapter 4 utilizes the high-level design to present a software design that can be readily implemented to build the framework. Finally, Chapter 5 presents a demonstration of the framework with several example systems to illustrate seamless integration of VR and AR across the Virtuality-Reality spectrum.

CHAPTER 2

BACKGROUND

This chapter establishes key concepts and background information necessary to understand the framework design and associated implementation. We begin by introducing the concept of an autonomous system and how it is modeled in this thesis to facilitate the research in this paper. This is followed by a discussion on the virtuality-reality spectrum and how it applies to Test & Evaluation of autonomous systems. Next, the Publish-Subscribe (PS) pattern is presented as a possible method for communication within the system to achieve flexibility and decoupling within a framework. The Robotic Operating System (ROS) is provided as an example of a PS system that is available and well-documented. Arduino programming and the Arduino IDE are also described to introduce a method of communication with physical hardware.

2.1 A General Model for an Autonomous System

It is important to understand the features of an autonomous system and how they interact before placing the system within a virtual or augmented reality setting. Autonomous systems require the ability to utilize various sensors to gain information about the external environment. Autonomous systems must also be able to interface with a physical system's actuators to instruct the system to act. Finally, control systems must be robust enough to adapt to changes in the environment, maintaining consistent feedback and behaving sensibly to a wide variety of possible situations [12]. To this end, the purpose of the control system is to generate a plan based on knowledge of the external environment and execute the plan based on actions made available by hardware actuators.

The modeling of autonomous control systems for mobile robots has an extensive history of research and development. One approach is to model the control system software using a pipeline of functional modules where input and output are connected to the robotic hardware [13], as illustrated in Fig. 1. The model includes, at a high-level, the autonomous software, the hardware, and the external environment. The hardware can be broken down into sensors, actuators, and vehicle dynamics/state information. The vehicle dynamics and state information include physical attributes of the vehicle such as velocity, orientation, and fuel level. The software can be further decomposed into the functional modules of the pipeline. The modules are:

- Sense – Computes a perception of the environment based on incoming raw data from the hardware sensors. This involves mapping the raw sensor data to the world representation.
- Plan – Generates a plan composed of actions based on the system's current world representation, operational goals, and past experiences.
- Act – Executes the plan by converting actions to control signals to send to the actuators.

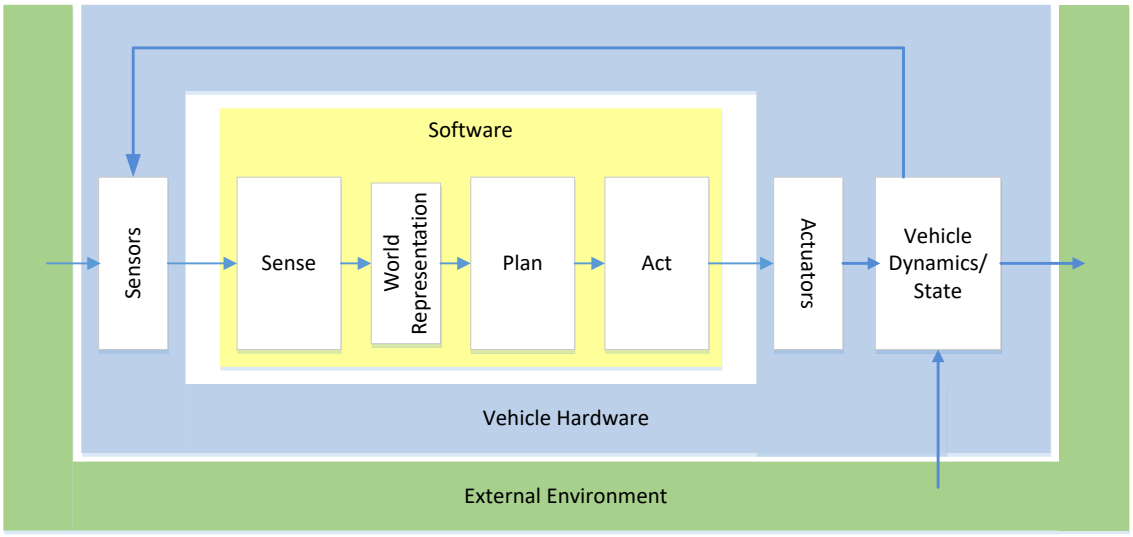

Fig. 1. Autonomous System Model.

The software also contains a world representation, an internal representation of the external environment and internal vehicle state. The world representation could include a panoramic view of distances to boundaries, sets of recognized objects and their computed attributes, or a map of the environment based on past experiences of the robot. Note this model is employed illustrate the ensuring work. The work is not solely relegated to this model and may include other stage decompositions such as to include a perception stage.

2.2 The Virtuality-Reality Spectrum

Properly utilizing virtual reality and augmented reality requires understanding the larger class of mixed reality displays. Milgram describes mixed reality displays as existing in a *virtuality-reality spectrum* [14]. During the design process, four phases of the reality-virtuality spectrum are introduced. This spectrum is bounded on one end by *reality* and on the other end by *virtuality*. In between these end points live a spectrum of mixed reality displays that includes *augmented reality* displays and *augmented virtuality* displays.

Davis and Lane provide an example of applying the virtuality-reality spectrum to design and testing of underwater vehicles [11]. A mixed reality framework is developed using JavaBeans and Java3D to model the environment and interface with physical vehicle hardware and software via an Ethernet-based communication network. The framework utilizes a communication protocol, OpenSHELL, to allow software modules within the framework to run remotely, allowing for reconfigurability across the virtuality-reality spectrum [11]. The conclusions show that extendable architectures can be developed and generalized to the testing of different autonomous systems. While only applied to underwater vehicles, the concept encourages further research in the area of autonomous system Test & Evaluation, supporting the need for the framework presented in this thesis.

Fig. 2 shows how the testing environment changes as the environment moves from a fully virtual reality to fully physical reality. Within the spectrum, a subset of components may operate in a simulated environment or augmented environment (involving both real and virtual stimuli), and the remaining components operate in a physical environment.

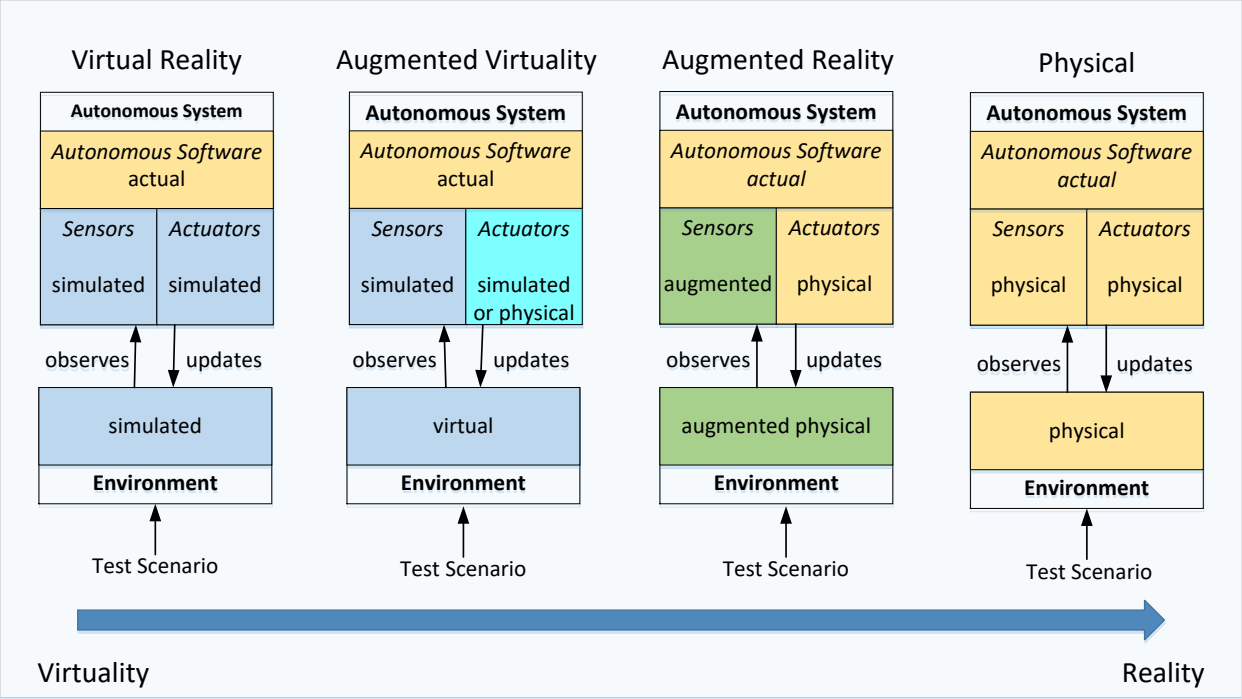

Fig. 2. Virtuality-reality spectrum.

At one end of the spectrum, virtual models of robotic components and various environments can be utilized. Note that while the hardware system is fully simulated, the current state of the autonomous software being tested is not a simulated version. The current state can progress from behavioral to algorithmic to functional as defined by the autonomous software development lifecycle. The simulation of the physical platform progresses from a behavioral model to a fully functional model as the specifications and design of the system progress, allowing greater and greater detail in the testing process.

As the design continues, more and more physical components are prototyped in software or hardware and used to augment the virtual system representation. This begins by allowing the physical robot to maneuver in the virtual world. All sensed information is provided from the virtual environment. The autonomous software can react to this information and physically

move the vehicle. By maintaining an avatar in the virtual world to represent the vehicle's state information in the physical world (position, etc.), the virtual environment can be appropriately sensed. This provides a safe environment to observe the vehicles response to various scenarios represented in the virtual world without risk of injury to people, the environment, or the vehicle itself.

In the final stages of design and testing, the autonomous software and vehicle hardware are complete and tested in the real environment. The vehicle now fully senses its physical environment and responds to it. However, for safety reasons it may be undesirable to place all objects in the real environment. For instance, people or other autonomous vehicles may be represented in virtual reality and then imposed on the real environment. Now virtual information is imposed on the real sensed information, requiring a stage prior to the sense or plan stages where real information can be augmented.

Testing may not end even after the robot is fully deployed in the real world. With the physical robot and autonomous software fielded, the autonomous software may need to be re-evaluated as the platform evolves. For example, this can occur if the software changes via updates to bugs or fixes for functional, performance, or security problems. It can also occur if the parameters in the software change by design over time, such as in deep learning models. In these cases, it is beneficial to allow the software to be placed back in the same operating environment used in the virtual or augmented stages of testing.

2.3 The Publish-Subscribe Pattern

A method of communication between system components is required that is flexible and maintainable among many different testing scenarios. Autonomous software design can introduce a complex web of dependences between components of the system making it difficult

to re-use or maintain system components along the virtuality-reality spectrum [15]. Decoupling is the process of reducing dependencies between software modules. A common pattern used to achieve decoupling in autonomous software is the Publish-Subscribe (PS) pattern [15]. Publish-Subscribe is a loosely coupled, message-oriented pattern for communicating in a network. The pattern has been used extensively in framework architectures for autonomous systems. These architectures focus on maintainability, performance, testability, extensibility [16].

The Publish-Subscribe (PS) pattern has four common types of message routing semantics: content-based, header-based, topic-based, and type-based [17]. Content-based and header-based systems route messages based on filters of either the message content or message header. Topic-based involve channels of which messages must match with a requested topic name to be routed. Type-based systems allows the selection of messages based on a selected type in a type hierarchy. For this discussion and further in the paper, we limit the scope of PS to topic-based systems.

Topic-based PS systems are primarily composed of two types of entities known as nodes and topics. Fig. 3 provides an illustration with an example of a camera publishing image data to two subscribers. Nodes are defined as a consumer and/or producer of information (i.e. a subscriber or publisher, respectively). Topics are defined as logical channels that associate a type of data to the channel such that nodes can communicate by interacting with the topic rather than directly with each other. When a node publishes information, it is to a named topic that accepts the type of content the publisher provides. A node that subscribes to information indicates a named topic from which to receive data and registers to be notified as information is made available through the topic.

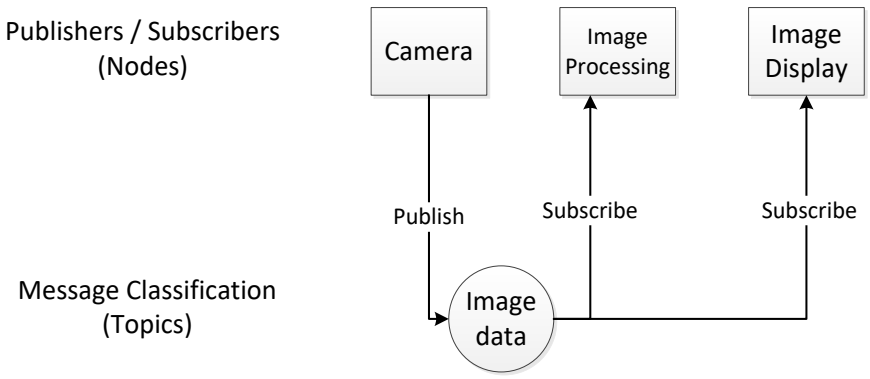

Fig. 3. Example Publish-Subscribe System.

Publish-Subscribe systems have a number of properties that are advantageous to connecting software components while maintaining flexibility and decoupling. PS systems exhibit space, time, and synchronization decoupling due to event notification [18]. Event notification is a method of delivering messages to subscribers in an asynchronous manner where the subscriber is notified when a message must be processed [17]. Memory space does not have to be shared between entities in the system allowing different parts of the system to exist on different platforms and be replaceable and interchangeable. Notifications can occur at a time different to when messages are sent or received, meaning that individual publishers and subscribers do not have to wait for messages to arrive to continue processing [18]. Synchronization decoupling also has the advantage of making the PS system more scalable. Nodes can be added or removed without directly impacting the performance of other nodes in the system [17].

2.4 Utilizing ROS for Framework Communication

The Robotic Operating System (ROS) is a collection of libraries that provide a Publish-Subscribe service on top of the standard inter-process and network communication layers of a

computer system [19]. ROS implements a topic-based Publish-Subscribe (PS) system between nodes in a peer-to-peer network. ROS nodes can be either on the same machine or different machines. ROS maintains its own interface for publishing and subscribing to specific topics.

ROS software is organized in groups called *packages*. The packages are divided into separate directories that may contain source code, third-party libraries, datasets, configuration files, and build files for compiling the nodes into runnable programs. There also exist several distributions of ROS available for development. Each distribution is a versioned set of ROS packages with a stable codebase from which to develop new software. Each distribution retains the same basic software architecture. The distribution used in this demonstration is ROS Kinetic which is typically used with the Ubuntu 16.04 Linux operating system.

The primary functions leveraged in ROS are subscription, advertisement, publication, and callback. Fig. 4 illustrates the processes at work within a ROS network. Subscription and advertisement are calls made to ROS in order to establish topics. Publication is a call made to ROS to initiate sending a message through an establish publication topic. Callback is a call made from ROS to the node upon receiving a message from an established subscription topic.

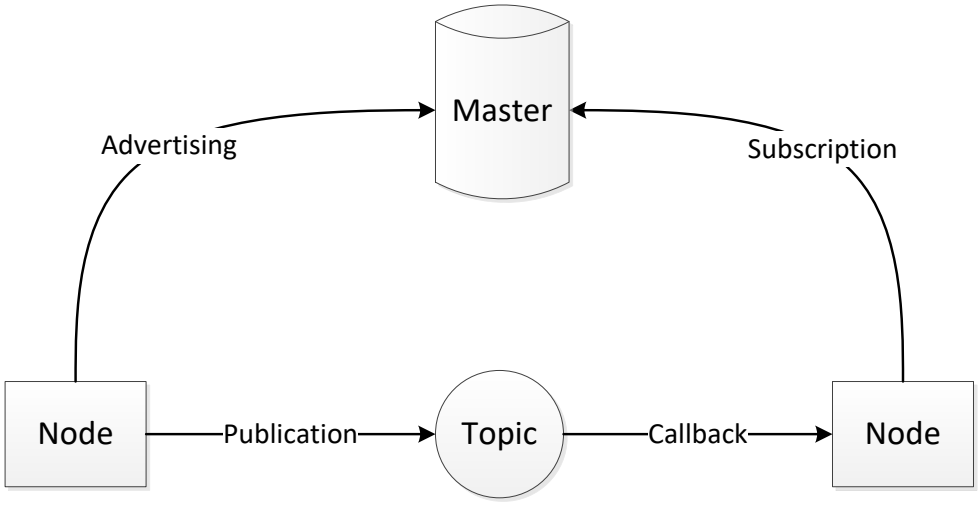

Fig. 4. Processes within ROS Communication.

These functions work in tangent with a server known as the Master. The role of the Master is to enable individual ROS nodes to locate one another to perform peer-to-peer communication [20]. Each ROS node registers with the Master that is started initially. Topics are then registered using a unique topic name and tracked with the ROS Master to provide information necessary for communication. Callback functions are registered by providing the function as a parameter to the call to subscribe to a given topic.

2.5 Utilizing Arduino for Hardware Communication

Arduino is a relatively easy-to-use and inexpensive platform for communicating with analog and digital devices that is programmable and extendable [21]. Arduino circuit boards come in a number of variants with differences in size, memory capacity, the number and type of connector pins, power, and processor speed. Arduino is utilized many times in academics and prototyping to demonstrate proof-of-concept robotics and sensing systems [21].

The Arduino IDE is a free development environment for developing programs that can be compiled, uploaded, and then executed on an Arduino board to interface with devices and control the Arduino's behavior. The programs are developed in source code files known as sketches. The IDE provides many features to accommodate development and testing of sketches such as error detection and serial monitor to help with debugging. There is also a large user community that publishes sketches for many commonly used devices, thus reducing development time.

Device communication can be handled by reading and writing in software to different input and output pins that are physically wired to pins on the devices. Libraries come with many devices with code that is already set up to interface correctly. Example sketches are also often available with each library to demonstrate use [22].

CHAPTER 3

AUTONOMOUS SYSTEM FRAMEWORK DESIGN

This chapter details the design and structure of the framework used to perform Test & Evaluation of the autonomous system software. The primary purpose of the framework is first discussed in relation to the virtuality-reality spectrum. A set of framework requirements are established to lay the groundwork for the rest of the section. A high-level view of the framework system and its components is then described. This is followed by the description of an example application that can be used for discussing the design. The framework is then decomposed to illustrate its basic components and interfaces to facilitate communication and component decoupling. This includes identifying and describing common components used within the framework. The component-type architecture is then mapped in terms of a Publish-Subscribe system and certain relationships are highlighted. Finally, the process of Test & Evaluation is described in the context of the framework. Key developer roles are discussed and put into the context of developing and testing the autonomous system using the framework.

3.1 Framework Purpose

The purpose of the framework is to decouple the components of the autonomous software from its operating environment and allow the software to interface with both virtual and physical components without direct knowledge. Decoupled, the software can be tested under different conditions without modification for each combination of virtual and physical components. At a minimum, the framework must support Test & Evaluation in the following scenarios:

- in a completely simulated environment with all components represented in virtual reality

- in a semi-simulated environment with a portion of physical sensors or physical actuators
- in a mostly physical environment with a portion of components in virtual reality
- in a completely physical environment with fully integrated sensors and actuators

3.2 Framework Requirements

At a broad level, the requirements for the framework facilitate building an effective test harness for the autonomous software. This includes the:

- ability to decouple components of the autonomous system for testing purposes
- ability to decouple the implementation of the autonomous software from its' operating environment, and the
- ability to decouple the autonomous system from knowledge of the source or use of information

It is important that the autonomous system components being tested are decoupled from each other. Test harnesses are efficient at a unit testing level where the function and performance of individual components can be specifically assessed and then later tested as an integrated whole. If the components are intertwined, it is more difficult to gauge individual component functionality. This characteristic is also critical for tracking down the source of problems in the system when they arise. For example, the reason behind a decision in the Plan stage could be difficult to determine if the input to the Plan stage is inaccessible. This could happen if the planning component(s) of the autonomous system are intertwined with the sensing component(s) of the system. Additionally, the input or output of certain components may be stochastic in nature. As such, an ability to inject or capture information to and from each component is essential to facilitating the testing process.

An effective test harness also decouples the implementation of the components from their operating environment. The operating environment consists of the system(s) external to the autonomous software that are available to provide input to the autonomous software and accept output from it. Decoupling the operating environment allows it to be substituted or replaced without modifying the autonomous software. This includes reducing the dependencies between the software Test & Evaluation and the ongoing development of the physical hardware. In this way, both the autonomous software and the external components can be developed separately without introducing testing dependencies into the project. In addition, decoupled components are easier to interchange or replace with different components later in the development lifecycle, adding fidelity to the testing. For example, a large-scale traffic simulation that is still early in development could be substituted for a simpler simulation that could fulfill the needs of testing while development continued.

Finally, to ensure changes in the framework do not affect the structure or design of the autonomous system, the autonomous software should be unaware of the source or use of information transferred through the system. This implies a need for formally defined data interfaces that remain consistent even if the data sources or data recipients change. For each component, input and output functions must be identified to connect the component with the rest of the system. These functions must be consistent between each component even if internal functions may vary.

3.4 Base Example

To build a foundation for later chapters, an example application will be used to provide context to the design. The application will be limited to the Sense stage to focus on the framework's ability to tackle decoupling of a single stage of the autonomous software. One

common application involving the autonomous software's Sense stage is the mapping of a robot's surrounding environment. The process involves using observations taken from sensors to build a world representation that can be displayed visually. The sensors include a proximity sensor and a compass sensor which are used to obtain the distance to obstructions within the environment and the robot's orientation within the environment, respectively.

3.3 General Approach

The focus of the framework architecture is to decouple the autonomous software components from their respective input and output sources and allow for additional components to control the virtuality-reality of the data without the autonomous software's knowledge. Fig. 5 illustrates a high-level view of the Test & Evaluation structure. Four major sections are highlighted. They are the:

- Physical (or simulated) Vehicle
- Physical Environment
- Virtual Environment and Test Scenario
- (Autonomous) Software
- Test Harness

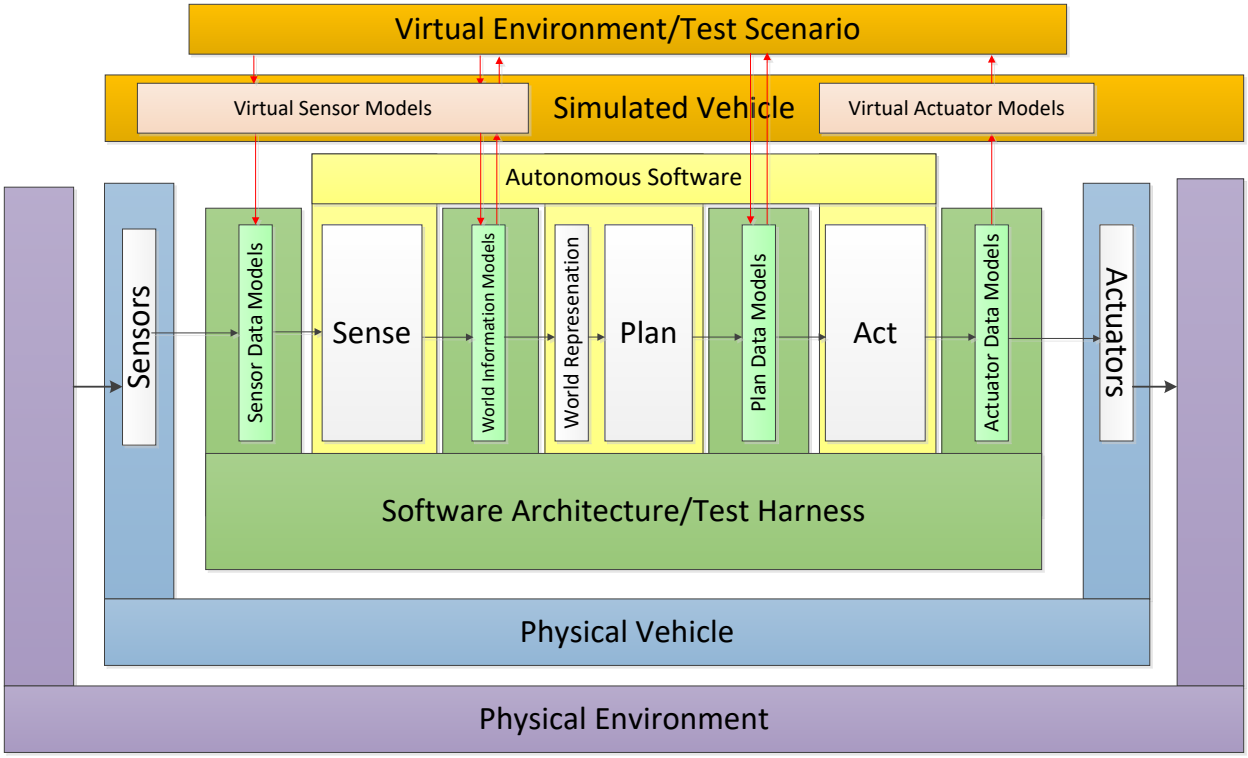

Fig. 5. Test & Evaluation Architecture.

The *Physical Vehicle* represents the system(s) that feeds information into the autonomous software and that uses information from the autonomous software to operate. It can be described as the body to the autonomous software's brain. It is composed of *Sensors* (which provide information to the autonomous software) and *Actuators* (which accept information as control signals to alter the vehicle's operation). The *Physical Environment* represents all the external factors and stimuli that can influence and be influenced by the physical vehicle. Information flow here is based on real-life interactions with a real system. The interactions are mainly with the physical *Sensors*, where information flows from the *Physical Environment*, or the physical *Actuators*, where change is enacted on the *Physical Environment*.

On the other side, the virtual systems represent generated or simulated versions of the physical counterparts. The *Simulated Vehicle* is composed of *Virtual Sensor Models* and *Virtual*

Actuator Models that represent the behavior of the physical counterparts found in the *Physical Vehicle*. The virtual environment, in general, represents a generated version of the environment external to the vehicle. This could be a simulation of the environment or simply a testing module(s) that provides a rough approximation of data obtained from the environment.

The components of *Sense*, *Plan* (including *World Representation*), and *Act* represent the main stages of the autonomous system model. They are separated based on the level of cohesion and functionality appropriate for each stage in the model. Each of these components has basic behavior defined according to the general model but may vary based on the specifics of the autonomous system. In general, it is understood that *Sense* transforms environment data into an appropriate form for the *World Representation*. *Plan* uses the information in *World Representation* to assess the environment and decide on a set of actions the system should perform. *Act* receives these actions and converts them into signals that the *Physical* (or *Simulated*) *Vehicle* can understand.

The *Test Harness* is placed in between the autonomous system components and the rest of the framework. This works to decouple the autonomous system from direct knowledge of the source and use of the information outside of each component's own processing. The *Test Harness* also allows for data manipulation. As the *Test Harness* handles the routing of information between different components, it also has access to the data before transfer. At the interface between autonomous software stages, the test harness provides a location where different operations can be placed to alter the data before transferring to one of the autonomous system components. This allows data to be injected or observed from outside of the autonomous software stages. It also allows the implementation of the virtual system(s) to change or be replaced if necessary, provided the communication format stays consistent.

Underlying the *Test Harness* is a communication layer that is able to connect the separate components of the framework architecture together. For our purposes, the Robotic Operating System (ROS) is this intermediate layer and provides much of the facilities for the type of communication that the framework requires. How ROS is integrated within the framework is discussed in more detail in Chapter 4.

3.5 Framework Decomposition

The framework system is now decomposed to illustrate its inner structure. The section first decomposes the high-level design into basic components that form the framework structure. Major groups for the framework structure are highlighted in context with the high-level design with focus on the group making up the *Test Harness*. Next, the framework is decomposed further to highlight certain classes of entities with different responsibilities within the *Test Harness* group. This is followed by a discussion of the decomposition within the sensing and mapping application.

3.5.1 Framework Components

The framework can fundamentally be decomposed into basic components known as nodes. Nodes are defined as a functional component of the framework that can produce and consume information. The nodes of the framework can be grouped into three main categories:

- Autonomous Software Nodes – *Sense, Plan, Act*
- External Nodes – *Virtual Environment, Virtual Sensor Models, Virtual Actuator Models*, and their physical counterparts
- Framework Nodes – *Sensor Data Models, World Information Models, Plan Data Models,* and *Actuator Data Models*

The Autonomous Software nodes involve the stages of the autonomous system model: *Sense, Plan, Act*. Each node has assumed behavior and responsibilities for its place in the framework architecture. Note that each node (*Sense, Plan, and Act*) may, itself, be made of multiple components that together, conceptually, work as a single node.

An external node refers to any component of the framework that exists outside of the autonomous software but is still able to influence or respond to the autonomous software's input and output. This includes nodes that interact directly with physical hardware or a virtual simulation of the hardware. It also can also include nodes that perform visualization based on observing the state of the autonomous software nodes or other external nodes.

Framework nodes provide the ability to choose from multiple sources of information external to the autonomous software nodes. These make up the *Test Harness* potion of the high-level architecture. As there can be multiple flows of information between the autonomous software stages, it is useful to view the test harness not as a single entity but composed of a variety of data models that operate independently for each type of data. A node is created for each type of data being communicated to control its flow.

The framework nodes can be divided into three sections, each preceding a stage of the autonomous software (*Sense*, *Plan*, and *Act*). To illustrate the decomposition, Fig. 6 shows only two sections specifically associated with the *Sense* node of the autonomous software. Each section operates independently on a subset of the data communicated between autonomous software stages and between the stages and external nodes (e.g. *Virtual Environment*). Furthermore, each data model independently acts on a single type of data within the subset.

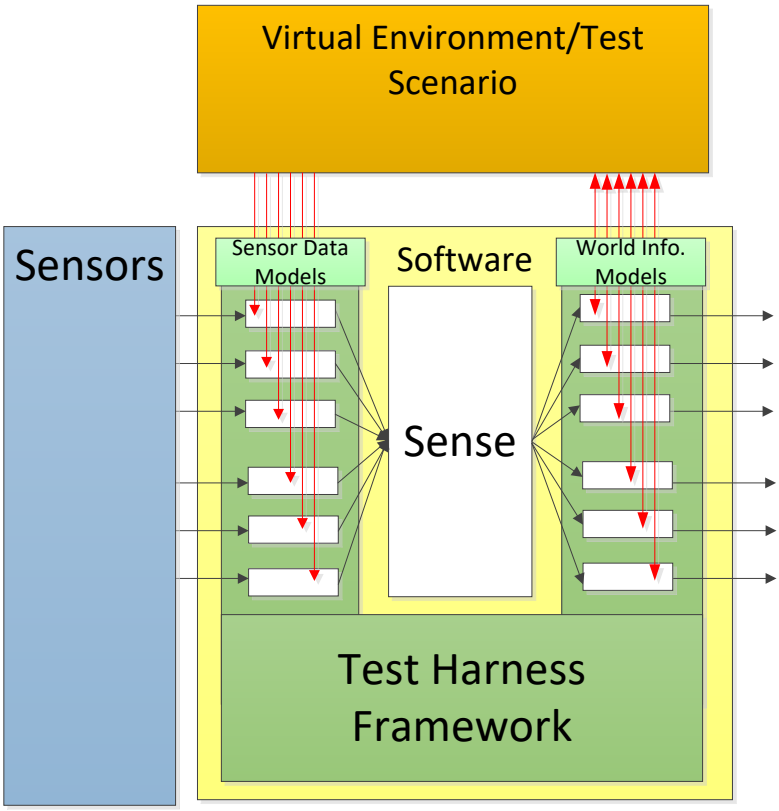

Fig. 6. Framework decomposition associated with *Sense* node.

As autonomous systems can have different information requirements, the decomposition of the framework depends largely on the needs of the autonomous system. Only data models that involve information that is actively passed into and out of the autonomous software should be present within the framework. Therefore, the framework structure is dynamic relative to the testing process and may need to change to accommodate the requirements for reducing dependencies in knowledge and implementation for certain pieces of information transferred through the system.

3.5.2 Framework Data Model Templates

Where the framework could contain numerous data models for various pieces of information travelling through the system, we find that three templates cover most models. These templates can act as basic data models for modeling the source or use of any type of information and could be further customized based on the specific data to be handled. These three classes and their mappings are:

- Combiner node (Many-to-1 mapping)
- Splitter node (1-to-Many mapping)
- Single-Valued node (1-to-1 mapping)

The framework does not preclude a many-to-many mapping, but we have not found the need for this template at the current time.

The combiner model involves multiple sources of information. As illustrated in Fig. 7, the source(s) of information are selectively outputted or augmented ("combination") from all sources of data. This generally involves only two sources: a physical source (or source from a previous stage of the autonomous model) and a virtual source. The combiner then has different *reality modes* to dictate the reality that is output as final data. If the reality mode is "physical", the combiner outputs the physical data; likewise, if the reality mode is "virtual", the output is the virtual data. The augmented reality mode requires modeling the interaction of virtual and physical data and may vary depending on the type of data being handled.

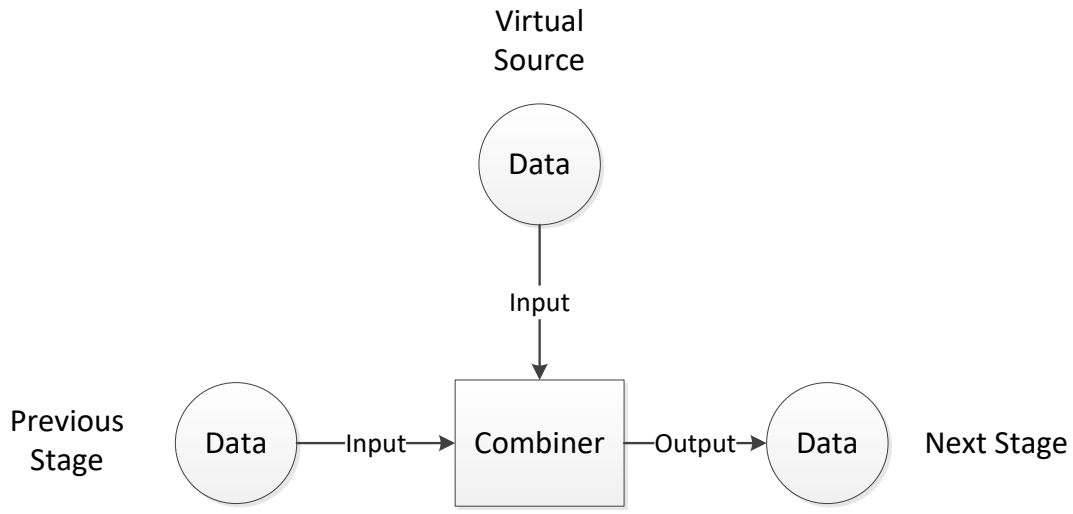

Fig. 7. Combiner Model.

A splitter model is used to pass forward data from a previous stage to a potentially multiple destinations, generally up to two destinations. Fig. 8 illustrates the splitter model. The splitter may operate as switch to allow output only to one of the destinations or a router to both at the same time. An example may include a model for routing control data from the autonomous model to physical and virtual actuators. Splitter nodes may also include additional modeling to transform data to formats appropriate for the virtual destination; for example, converting data to an appropriate coordinate system, which may differ between the virtual and real worlds.

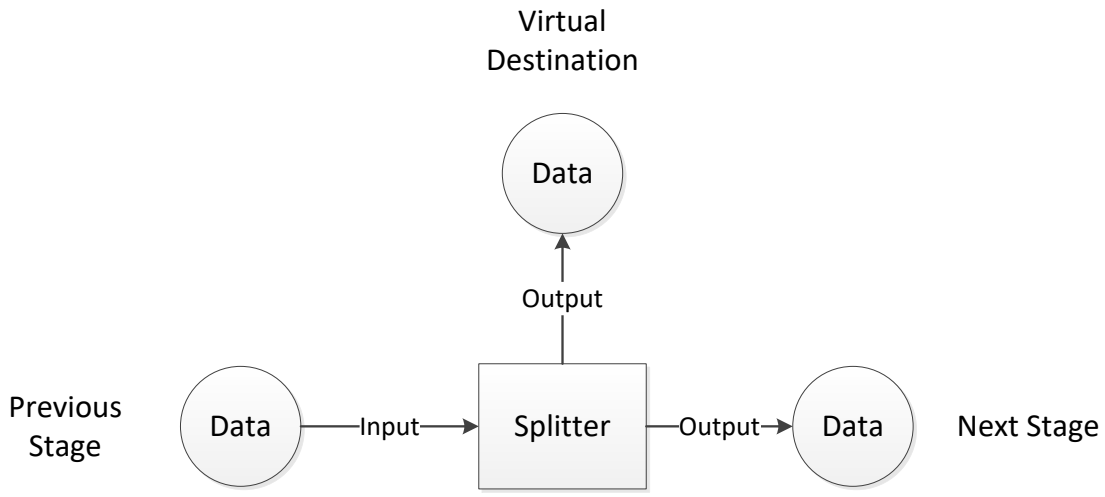

Fig. 8. Splitter Model.

A single-valued model is used to output data from a single source, such as shown in Fig. 9. It may still include modeling to process data before output. For example, the data may need to be converted to a certain representation or coordinate system for testing purposes. Another example may be thresholding values or passing averaged values to smooth data and eliminate or add noise, though it is advocated that such functionality probably is not appropriate in the framework. In general, the single-valued model should not include modeling defined by the autonomous system.

Fig. 9. Single-Valued Model.

3.5.3 Base Example Decomposition

To illustrate these classes, the base example is mapped to the framework. There are two types of observations associated with the base example: range data and orientation data. The framework is then decomposed into two corresponding data models. These two models are placed in between the autonomous software and external components. In other words, any input for range data or orientation data must go through the combiner and be made accessible to the framework before being inputted into the autonomous software.

The range finder may use a combiner to control the perception of range data. Two sources are made available to the combiner: the range data from the physical sensor and range data that is generated from a virtual environment. Fig. 10 shows that the flow of information is altered by the reality mode. If mode is for "physical" or "virtual", only one of the inputs are used to determine output. If by augmentation, the output is determined as a function of both inputs. Additionally, a threshold may be applied to the output to only consider range values no closer or no further than the threshold for testing.

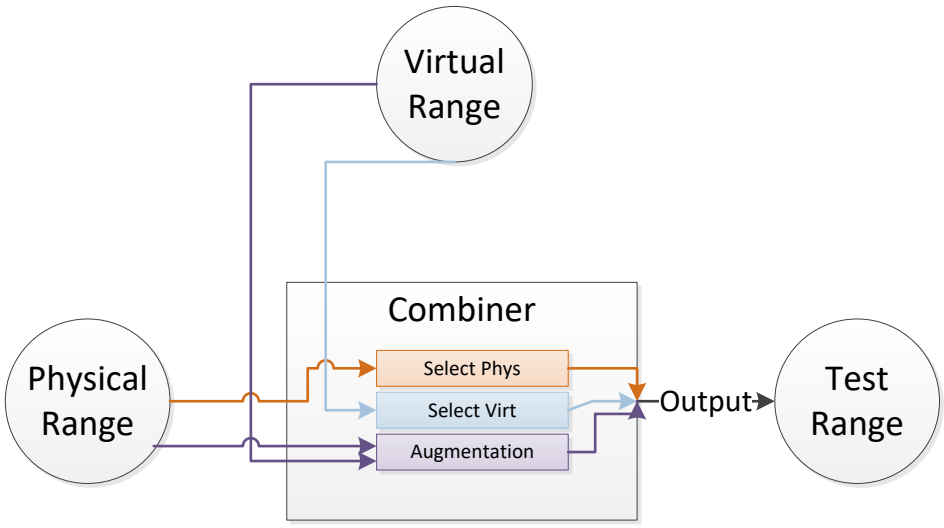

Fig. 10. Information flow in an example Combiner.

3.6 Node Communication

Isolating knowledge between the nodes within the framework is now considered. Interfaces between decoupled nodes must remain consistent for nodes to communicate accurately and predictably. For example, suppose there is a transportation system with two components: a traffic simulation and an autonomous driver. The transportation system provides information about the presence of pedestrians in the simulated environment, and the driver can accept this information about the environment and provides information about whether the driver's vehicle should brake or turn. How the traffic simulation produces the information is subject to change during system development based on the fidelity of the simulation required; however, the simulation must still provide appropriate information for the driver to be accurately tested.

Communication can be hidden by interacting through a channel. This channel is similar to the concept of a topic. The main characteristic of this interface is that components do not directly communicate with each other (including those components in the *Test Harness*). Each node that produces information will push the information through the channel. In addition, nodes that need to consume the information will reference the same topic. As Fig. 11 shows, from the perspective of each component behind the interface, the actual producer or consumer is not known. The channel is known, but the nodes do not have specific knowledge of the information medium or method of communication, only that certain information can be sent or received going through the channel.

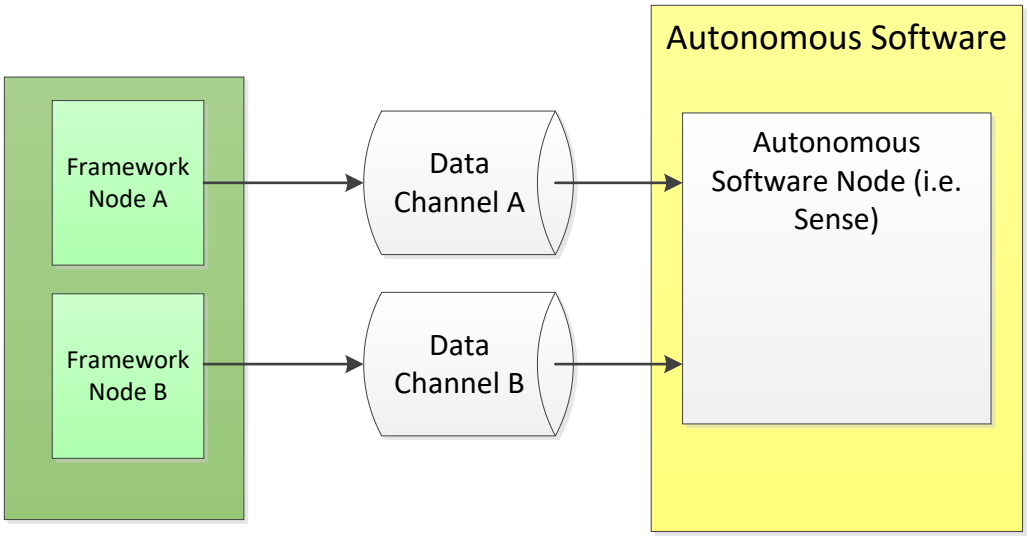

Fig. 11. Communication via channels.

Consequently, this type of interface must adapt to the information requirements of each component. Each component has different inputs and outputs which may contain different types of information for testing. For example, if a Sense node requires range data as input, the associated interface must have knowledge of the channel(s) that carries this data in order to properly receive the range data in isolation. On the other hand, the interface for a node that provides orientation data requires knowledge of a different channel beforehand to identify the channel for orientation. With this, we can outline basic requirements for each component to communicate through the interface:

- Nodes interact with the channel rather than directly with other nodes
- Channels do not change when nodes are replaced
- Channels are associated with a type of data of which the format is known to be readable by other nodes

3.7 Mapping to a Publish-Subscribe Pattern

Underlying the Test Harness is a communication layer that connects the separate components of the framework architecture together. The communication layer primarily manages the flow of information from node to node without nodes having knowledge of specific senders or recipients. The paradigm that dictates this interaction is the Publish-Subscribe (PS) pattern. The pattern is a reasonable choice as it does not involve direct communication between sender and recipients (publishers and subscribers) and allows for network entities to be replaced without impacting the whole network [18].

In this way, mappings can be made from the architecture to the pattern. Each node in the framework can be mapped to one or more nodes of a Publish-Subscribe system. The concept of logical channels can be readily mapped to topics in a topic-based Publish-Subscribe system. The interfaces each node uses to communicate information becomes an interface to access a publish-subscribe service. Nodes for the physical sensors become publishers of sensor data in the network, where the actuators become subscribers. Other nodes are both publishers and subscribers, such as the data model(s) that make up the Test Harness.

Each node has a basic set of behaviors to communicate over the network. These include the ability to advertise, subscribe, publish messages, and receive messages through callback. In addition, each topic must have a well-defined and known format that is used by publishers and subscribers to understand the message content. This includes the ability to serialize and deserialize data fields into a common communication format.

To illustrate an example of the mapping, the base example is utilized, illustrated in Fig. 12. The main nodes involved here are the Sense node (component of the autonomous software), the Virtual Environment node, and two framework nodes: one representing a Combiner for range data and the other representing a single-valued node for orientation data. The arrows represent

the flow of information through the system once publishing and subscription have taken place. The arrows are labeled based on which process they refer to: publishing or subscription.

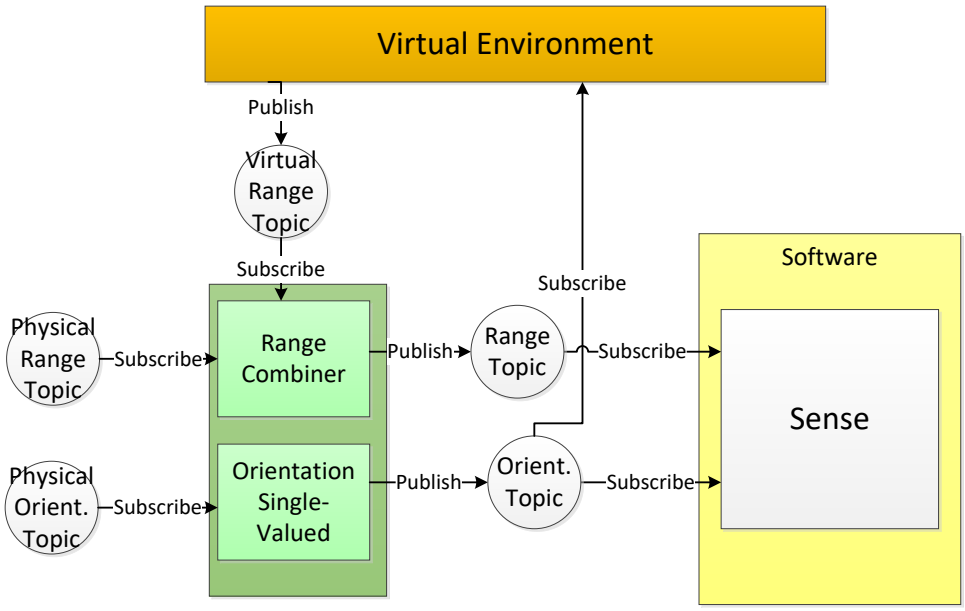

Fig. 12. Applying Publish-Subscribe to the Base Example.

Finally, the circles represent the several topics that channel a particular type of data from the *Test Harness* to the *Sense* node. This includes a physical and virtual range topic, a physical orientation topic, and two topics for the range and orientation input into the autonomous software.

The result is that the autonomous software is decoupled from the implementation of the virtual environment and physical sensors. Those nodes could be replaced if the communication channel is maintained in the form of a topic with known message content. Similarly, the virtual environment could be replaced without affecting the Range Combiner. In addition, the Sense node is decoupled from the source of the range and orientation data. This allows framework nodes such as the Range Combiner to perform data manipulation (through either selection,

augmentation, or some other process) on the range before it is published to the topic, as long as the topic format is consistent. This design can be very powerful as it inherits the flexibility and scalability that is attributed to the Publish-Subscribe system.

3.8 Developer Roles

In addition to the design of the framework, it is also important to discuss developer roles and responsibilities. The entire system hosts several components that may have their own lifecycle for design, development, and testing. Different developer roles exist to design and build this testing system. Knowing what roles exist can help in identifying who is responsible for developing and managing certain components and associated communication channels and organizing the project resources and timeline.

Key development roles are identified based on the decomposition of the architecture. These follow from the major components:

- Autonomous Software Developer
- Physical Vehicle Developer
- Virtual Vehicle Developer
- Virtual Environment Developer
- Framework Developer

It is important to note that each developer does not necessarily have to correspond to a single person; but rather indicates a developer entity that could be a team of engineers or even an organization. Each developer is responsible for one of the major components of the system. In addition, at any one time, these developers could be at a different point in their own development process. For example, the vehicle's hardware may still be in an early design phase at the same

time the autonomous software has begun testing. In addition, at this point, a virtual environment may be only rudimentary and static, with complex behavior and interactions within the environment still in development. At this point, the virtual vehicle may be a simple behavioral model to allow initial testing without the hardware. Framework developers must especially work to stay ahead of other development teams to facilitate testing of the different components and ensure accurate results. The disparity in development presents a challenge for testing the system in its entirety; until, at least, very late in the lifecycle.

The key to overcoming the design challenges is by leveraging the framework's flexibility to adapt the system testing to suit the current state of development. The Publish-Subscribe pattern and framework design allow for individual nodes to be replaced or added as necessary. However, this means the burden is placed on the communication channels (i.e. topics) to maintain a consistency between each node. Subsequently, the framework developer is the party responsible for maintaining the communication channels as they are responsible for developing the *Test Harness* that lies in between each major component. To achieve consistency, the framework developer must keep up-to-date with the necessary requirements for each node to properly configure the channels.

For example, consider the base example discussed previously. What if the autonomous software was updated to require the position of the autonomous vehicle in the environment? Fig. 13 shows a possible structure with updates.

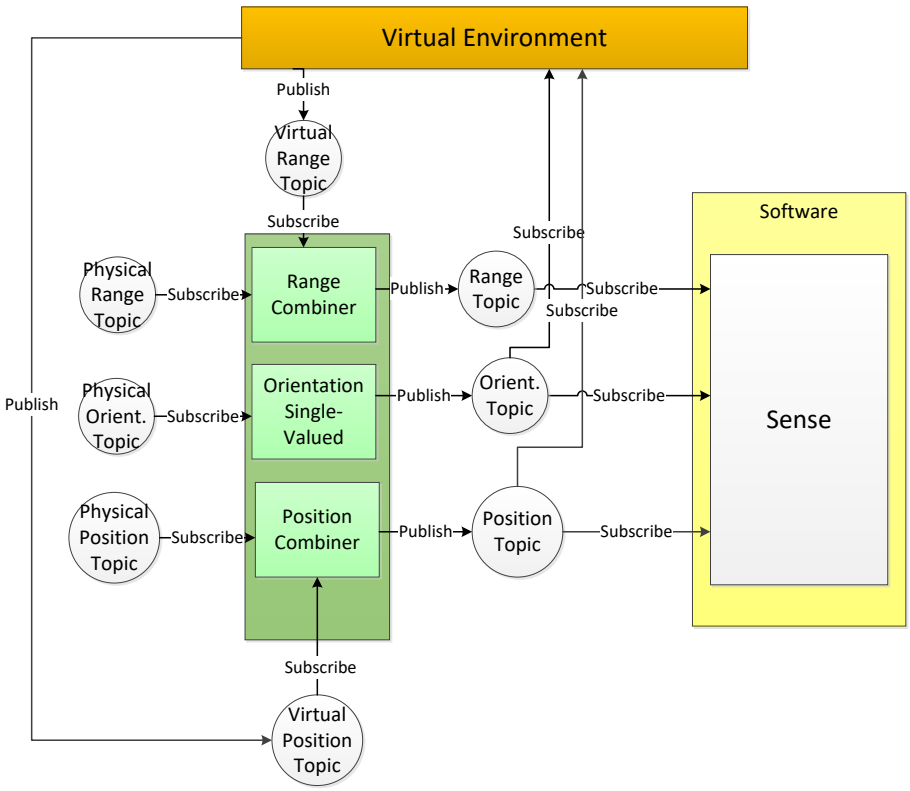

Fig. 13. Updated Base Example with added Position Sensor Data.

The autonomous software does not have this information readily available and must receive it from and external source. As such, a communication channel must be established by creating a topic for receiving the position data. The data would also need a defined format for communication. In addition, a framework node must be developed that captures the position data and makes it accessible for data manipulation (for selection, augmentation, etc.). Finally, at least one source of position data must be made available to publish data into the system (examples could include a physical sensor or a virtual simulated sensor). Other nodes such as the virtual environment might also subscribe to the position data for the purposes of synchronization. As such, their communication channels would also be affected.

CHAPTER 4

FRAMEWORK SOFTWARE DESIGN

This chapter discusses the details of the software design to support the framework. A set of requirements are derived from the general framework design developed in Chapter 3. A basic approach is outlined for the software design, to include the programming constructs that are relevant to developing nodes in the framework and major events that are captured within the framework. Details are described for the base classes that make up the foundation for the framework implementation. Finally, where and how ROS is utilized to implement the given design is discussed.

4.1 Software Requirements

To start, a set of requirements are derived for the software design from the framework design. These requirements are important for identifying and organizing required behaviors and developing an approach for the implementation. Requirements are:

- that necessary behaviors for communication are provided by a consistent software interface,
- the interface refers to data topics only and does not refer to any node specifically,
- the interface works independent of the type of node or type of data communicated,
- the details behind the interface can be easily replaced, for instance, with an alternate Publish-Subscribe service, and
- the interface is able to accommodate an arbitrary number of nodes and interconnections.

From the framework design, the framework is composed of functional components known as nodes and a communication layer that decouples each node from the source of input data or use of output data. An application programming interface (API) is defined to abstract the behaviors of a Publish-Subscribe system, allowing different nodes access to the PS communication layer. The application does not require knowledge of the details of the communication layer as long as the API remains consistent from node to node. In addition, the API standardizes the development of nodes in framework and establishes a specific set of rules and procedures for creating any node.

The API cannot refer to any node specifically as a target to send or receive data. Doing so would violate the framework's requirement to decouple the implementation of the autonomous software from its operating environment. Instead, each node must communicate through channels known as topics via the behaviors defined by the Publish & Subscribe pattern. The API is limited by these behaviors and must enforce them on the application to enable communication. This includes providing routines to declare publication or subscription given a topic name and message content/format. It also includes providing a way for the application to inform and be informed of the communication.

The API must also be independent of the type of node being developed or data being communicated. The process of sending and receiving messages assumes the format of the messages can be readily understood by both the sender and receiver. As the API abstracts the implementation of the communication, it must also abstract how the messages are viewed. Indeed, the message format used to communicate the message between nodes is likely not the same format that is used in the application. The contents of messages can also change based on the application. As such, the API must provide a way to specify the content for a given message

type. This includes how to map the data between the format used by the framework and the format used by the application.

The underlying communication layer, or backend, may handle communication differently depending on Publish-Subscribe service used. Different backends, such as ROS, AMQP, or MOOS-IVP, may have different processing requirements or conditions for use. In addition, a different implementation can be used at various stages of development. For example, an easy and robust commercial package such as ROS may be preferable early on in development of the framework. Later, when performance is more critical, a more efficient, custom solution may be required without altering the application(s) that have been developed. Therefore, the backend must be easily replaceable without modifying the individual nodes of the framework.

4.2 Design Approach

To meet these requirements, the API includes behaviors for communicating between decoupled nodes in a Publish-Subscribe system. Specifically, these are the:

- ability to create node within the framework
- ability to subscribe to a topic
- ability to advertise a topic
- ability to receive messages from a subscribed topic
- ability to send messages to a published topic
- ability to be notified when messages are received

The API behaviors are mapped to several functions listed in Table 1. The functions are placed in a rough order of precedence that they should be called to handle communication appropriately. The purpose of *Initialization* is to create the node and connect it to the framework

and allow specialized classes to perform their own initialization alongside further calls to connect to topics. *Subscribe* and *Advertise* are functions that connect a data object provided by the node to a certain topic name for either receiving or sending data along the topic when notified to do so. *Notification* is for connecting a function provided by the node to a certain (subscribed) topic to be called on the event of receiving data on the topic. *Callback* and *Publish* are specifically for receiving or sending data to and from a topic, respectively. This includes deserialization and serialization as part of the communication.

Table 1. API Functions.

Function	**Behavior**
Initialization	Creates node and performs node initialization
Advertise	Advertises a topic and connects data for sending to the topic
Subscribe	Subscribes to a topic and connects data to receive
Notification	Connects function to be notified upon receiving from a topic
Publish	Sends data connected to topic for publishing
Callback	Receives data connected to subscribed topic

The API is made accessible to the application through inheritance. Base classes implement the underlying abstraction of the Publish-Subscribe service, and specialized classes utilize the API to implement the node-specific functionality. Messages are also encapsulated as classes which include relevant fields and mapping through serialization/deserialization. As the API is made accessible through inheritance, this approach assumes that applications follow the proper rules for building specialized classes and implement virtual methods that are required for the system to operate.

Fig. 14 illustrates the inheritance relationship between the base classes and specialized classes and the visibility to the developer. The bottom portion shows the base classes from which nodes would be developed. The top portion shows the specialized or derived classes that would be unique to a specific node or specific type of message in the system. The declarations of the base classes are visible to node developers to create the specialized classes for those nodes, but the definition is hidden to allow for replaceability of the communication layer to a different Publish-Subscribe service.

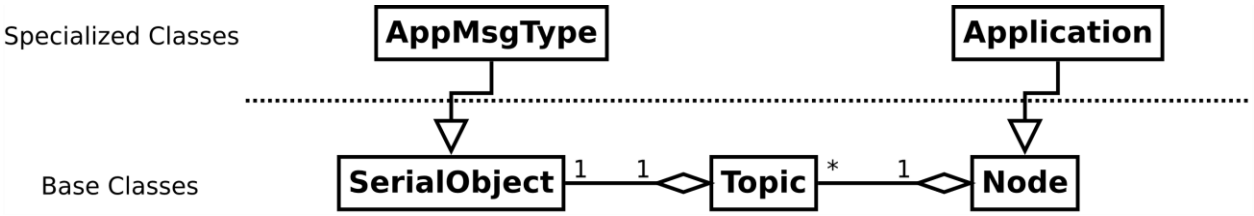

Fig. 14. Inheritance of *Application* classes.

The *Node* class encapsulates the behaviors for managing topics and methods for notification and processing. The purpose of the *Node* class is to automate the Publish-Subscribe communication while providing access to the application node to connect data objects and functions that define the node's state and behavior. Each *Node* publishes or subscribes to one or more topics which are represented by a *Topic* class. The *Topic* class encapsulates functionality necessary to send or receive from a single topic. Each *Topic* has reference to one other *SerialObject*. The *SerialObject* represents a base class for specifying message content and format for messages passed along a topic. The *SerialObject* also provides a mapping of data through serialization/deserialization.

In addition, the software design adopts the singleton pattern. The pattern involves establishing a single instance of the *Node* class. Other instances of *Node* are not allowed. This ensures that initialization for each application node is only performed once and defines a specific area of each application node that can appropriately access core functions and variables. This also means that each application node is built from within the derived *Application* class. Likewise, the base *Node* class does not have specific behaviors defined for it and is not meant to be used without inheritance.

To better follow the terminology of later sections, it is important that we make a distinction between a node as a program and as a class. From this point, referring to "*Node*" or "*Application"* will refer to the class or derived classes that encapsulates the Publish-Subscribe interface and node specific behavior, respectively. Referring to "application node" or simply "node" will refer to the program that contains a specific *Application* class.

4.3 Implementation of the API

In this section, details are presented for how the API is organized and accessed from within different nodes. API functions are made available by inheriting an abstract class *Node* which defines the functions in terms of the underlying Publish-Subscribe system (e.g. ROS). A *Topic* class is used behind the API to organize the functions and information associated with a single topic. The data objects that can be provided to the API are inherited from the *SerialObject* class, allowing the developer to define the serialization process. Finally, two classes are explained relating to framework nodes that implement a general *Combiner* or *Splitter*. These classes provide an example of classes that derive from *Node* and can be further specialized to handle data within the virtuality-reality spectrum.

The base *Node* and *Topic* class contain declarations for API behaviors that may be implemented differently depending on the underlying communication system. These primarily include *Subscribe (), Publish (), Init (),* and *Loop ().* The specific definition of these functions may change when moving to another system such as AMQP or MOOS-IVP. These functions are declared as abstract so the communication can be replaced if necessary. However, the declarations and expected behavior is assumed not to change when moving from one system to another. Following this structure allows application nodes to continue working using the same methods when a using a different communication layer.

4.3.1 Node Class

Every node has an entry point (main) function. There are two procedures that the main function calls to run the node: initialization and the control loop. Initialization starts by creating the singleton instance of the *Node* class from which the rest of the application node is run. Specifically, a derived class of *Node* (or *Application* class) is created that implements specific behavior required for the node. Initialization then involves calling *Setup()* to allow the node to connect data and functions to topics.

The call to *Setup()* is important as it allows the *Application* class to customize the basic *Node* control loop and define the node's behavior. The basic control loop is shown in Fig. 15. Evidently, the loop is largely dependent on the functions connected or registered during Setup. For example, if Setup contained an empty definition, the node would have no functions to call; and, therefore, result in the node having no defined behavior.

The *Subscribe()* and *Publish()* methods allow the *Application* class to connect data objects to specific topic channels. Once connected, the current state of the data objects will be used whenever data is received (through a subscription topic) or whenever the *Application* class

indicates data should be sent (through a publication topic). *RegisterInputFunction()* is essentially the *Notification* function listed in the API and connects a method to a subscription topic. The method is automatically notified (called) on the event that new data is received from the associated topic.

```
foreach function in init functions:
        call init function()
while not terminated:
        process incoming and outgoing msgs
        foreach function in core functions:
                call core function()
        foreach topic in publishers:
                if topic flag is true:
                        call topic publish()
                        reset topic flag
foreach function in exit functions:
        call exit function()
```

Fig. 15. Control Loop Routine.

Similarly, *RegisterInitFunction(), RegisterCoreFunction(),* and *RegisterExitFunction(),* respectively, allow the *Application* class to connect other methods to the control loop; however, these methods are called in regular places in the control loop rather than on an event. An additional function *FindTopicName()* is included for retrieving a topic for a given parameter at runtime. The *Node* class functions are summarized in the class diagram shown in Fig. 16.

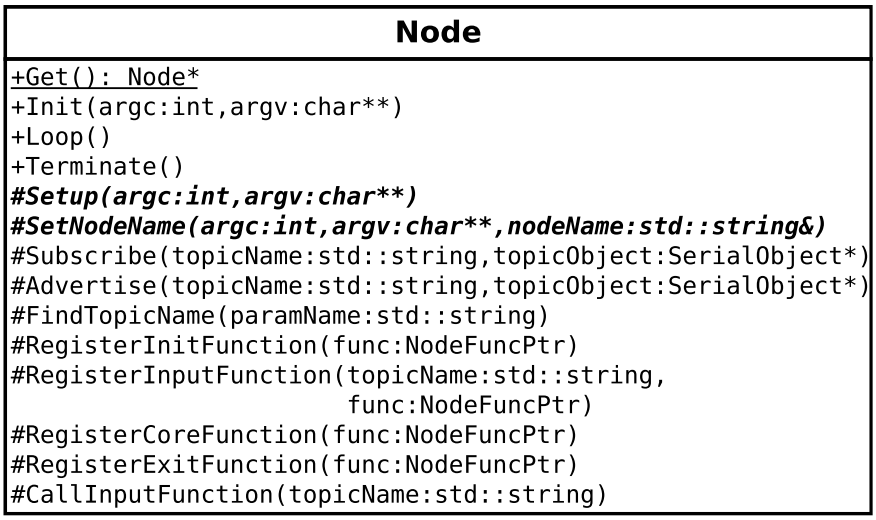

Fig. 16. Node Class Functions.

4.3.2 Topic Class

The *Topic* class encapsulates the functionality and information necessary to communicate along a single topic, shown in Fig. 17. This includes the *topic name*, a reference to the associated *SerialObject*, and any other data objects necessary to support the communication. It also contains the methods for sending information on a single published topic or receiving information from a single subscribed topic. The two main functions that it must implement are *Publish()* and *Callback()* to handle sending data to the topic or receiving data from the topic. Like the *Node* class, the definition of this class is not visible to the developer and may differ based on the communication layer.

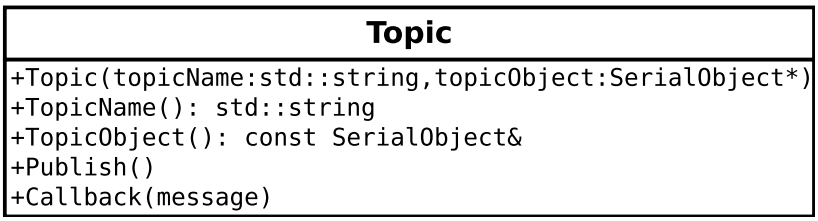

Fig. 17. Topic Class Functions.

4.3.3 Serial Object Class

As shown in Fig. 18, the *SerialObject* class is a base class for specifying data objects that can be connected via the API to topics for communication. Specialized classes would contain additional fields for message content. The base class also have three main methods which should be implemented in specialized classes. These are the *Serialization()*, *Deserialization()*, and *GetObjectSize()*. The former two methods are used to handle data mapping for sending and receiving. The *Serialize()* method takes the message fields and copies their current values into a provided byte-string. The *Deserialize()* method performs the opposite transformation and copies values from a provided byte-string into the correct message fields.

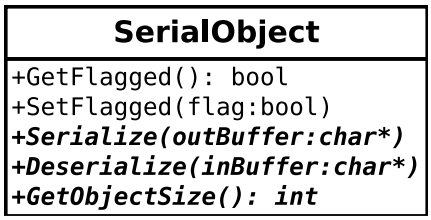

Fig. 18. SerialObject Class Functions.

The *SerialObject* assumes it will receive a byte-string of the correct size and with the appropriate format given the order of serialization. Each specialized class is also responsible for copying the appropriate fields in the correct order to and from the byte-string representation. *GetObjectSize()* should return the summed size in bytes of all the relevant message fields. For this implementation, it is also assumed that, for given *SerialObject*, this message size does not change during execution.

Additionally, two functions *GetFlagged()* and *SetFlagged()* are used with the *Topic*'s *Publish* function. *SetFlagged()* is used by the *Application* class to mark the *SerialObject* for serialization and trigger publishing on the associated topic. *GetFlagged()* is used by the associated *Topic* class to identify when to serialize and publish the associated *SerialObject* using the communication layer. The flagged state is then reset after publishing.

4.3.4 Combiner and Splitter Classes

On top of the basic *Node* class, two extensions of *Node* are the *Combiner* and *Splitter* classes. Fig. 19 illustrates where the two classes fall in the class hierarchy. These two classes inherit the base class functionality and extend it to enable selection or augmentation of the input and output to and from the autonomous software nodes. The *Combiner* and *Splitter* classes provide convenience by providing methods that can be reused and extended to manipulate different types of data (i.e. single value, 2D image, transform, etc.).

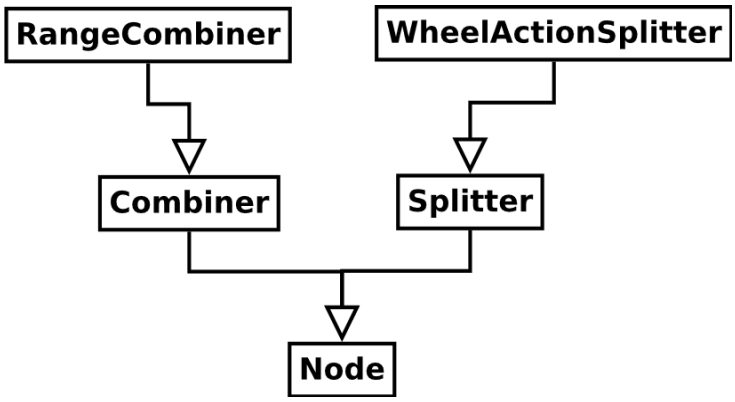

Fig. 19. Combiner and Splitter Hierarchy.

Table 2 shows the additional abstract functions made available in the *Combiner* and *Splitter* class that allow for customization while controlling behavior. *CreateObjects()* allows a

Combiner class to define and allocate memory for data objects that represent the input and output data for the *Combiner/Splitter*. Likewise, *SetTopicNames()* allows the node to define the names for input and output topics.

Table 2. *Combiner* and *Splitter* Abstract Functions.

Function	**Behavior**
CreateObjects	Allocates memory for data objects as specified for input and output data
SetTopicNames	Assigns topic names as specified for input and output topics
SetMode	Assigns the reality mode
Combine	Computes output data based on phys. and virt. data and reality mode
Split	Computes phys. and virt. data based on input data and reality mode

SetMode() allows the reality mode to be assigned based on parameters passed to the application node at execution. *Combine()* and *Split()* compute output of the *Combiner* or *Splitter*, respectively, based on the input data available and the assigned reality mode. The reality modes are defined as such:

- Mode 0: Select/Use only physical data
- Mode 1: Select/Use only virtual data
- Mode 2: Use augmentation of physical and virtual data

Both the *Combiner* and *Splitter* classes define *Setup()* to call the abstract methods and customize the topics and data objects. The core routine then calls the *Combine() or Split()* function as appropriate to perform selection or augmentation. Fig. 20 shows the *Combiner*'s core routine as an example. If a *Combiner* class' reality mode selects only virtual data, the *Combiner* does not need to subscribe to a physical data topic.

```
If mode is 0 AND physical received true:
        Call Combine() using physical data and return output
        Publish output data
        Set physical received to false
Else if mode is 1 AND virtual received true:
        Call Combine() using virtual data and return output
        Publish output data
        Set virtual received to false
Else if mode is 2 AND physical received true AND virtual received true:
        Call Combine() using physical and virtual data and return output
        Publish output data
        Set physical received to false
        Set virtual received to false
```

Fig. 20. *Combiner* Core Routine.

Similarly, the core function does not wait until physical data is available before calling to process the incoming data. As such, the reality mode alters the behavior of the core function.

4.4 Implementation Using ROS

ROS fills in the Publish-Subscribe functionality that is required by the API. Some of the functionality is at the code-level where API functions call on the ROS service to perform appropriate behaviors. ROS functionality is also at the build-level where application nodes are compiled into executable programs. Each node can be developed separately from the Publish-Subscribe system by utilizing the provided API. However, when using ROS as the communication layer, the nodes must be compiled within the ROS (or Arduino) environment and executed via ROS's runtime services to communicate properly. A ROS *launch file* is constructed to identify what nodes should be executed and what topic names are available for publish/subscription. In the future, the launch file could be replaced with an abstracted representation of the topology that could be mapped to ROS launch files or an equivalent mechanism provided by the utilized Publish-Subscribe system.

4.4.1 Code-Level

The API interacts with the ROS service primarily in the initialization and control loop methods. The application node first performs ROS initialization. This includes creating a ROS handle and calling a specific ROS initialization function to allow the ROS backend to allocate what it needs to function. With ROS initialized, calls can be made to subscribe and advertise to different topics. The method then calls the *Setup()* function to allow for the application node to perform the subscription and advertisement and connect data and functions to send, receive, and notify when appropriate.

The application node then moves to the control loop. Fig. 21 illustrates the interaction of framework's API with ROS's API through a sequence diagram. Control starts with *Node* and passes into the ROS API with a call to *spinOnce()*, which is ROS's function for processing incoming and outgoing messages. The framework API resumes control when an incoming message is passed to a callback function registered automatically with ROS when subscribing to the topic. The callback then deserializes/copies the incoming message into the appropriate *SerialObject* connected to the topic. It then notifies the appropriate input function if one is registered with the topic.

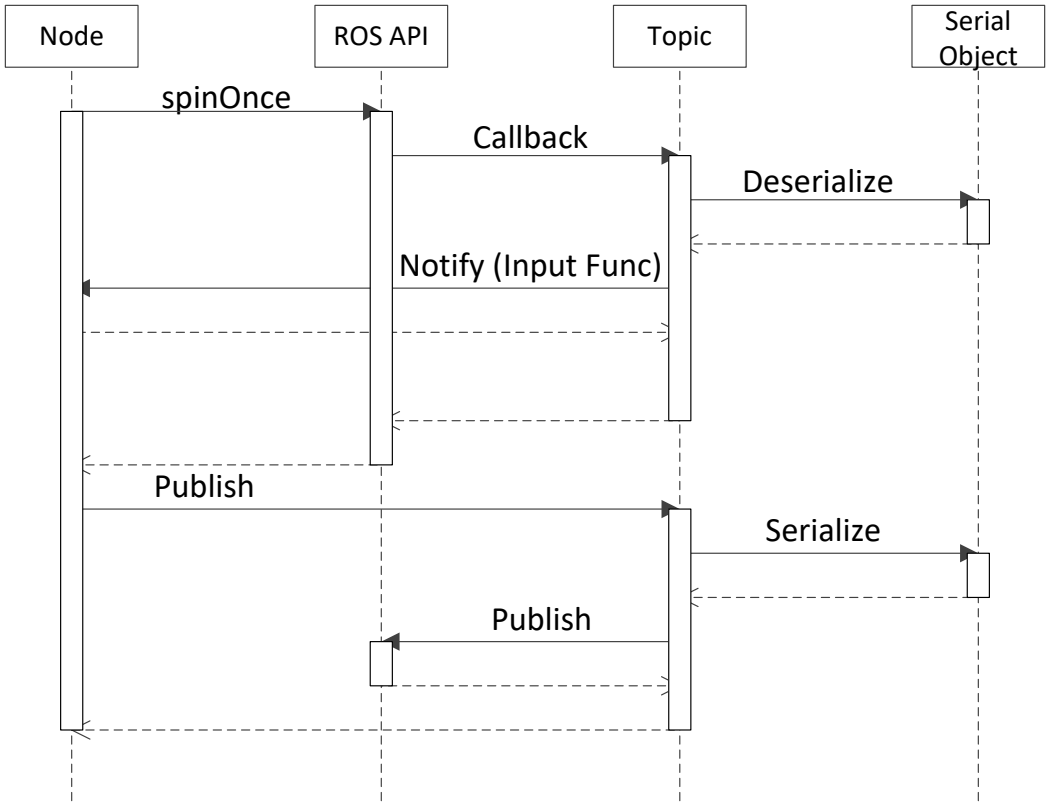

Fig. 21. ROS Interaction Sequence.

Later in the sequence, assuming a *SerialObject* is marked as flagged, *Node* calls the associated *Topic* to publish the data. This involves calling the associated *SerialObject*'s *Serialize()* to obtain serialized data to publish and then calling ROS's *Publish()* function to pass the data to the communication layer.

It is important to note that the ROS callback is not the same thing as the registered input function. ROS requires a callback to receive the message, but an input function for the application to be notified is not required. In a different manner, the call to publish from a topic first calls a routine to serialize the data from the connected *SerialObject* and then relays the data to the ROS API for its own processing.

Fig. 22 shows a more detailed view of where and how the behaviors provided by the API are connected to the underlying Publish-Subscribe service provided by ROS. The application

section contains *Input functions* and *Core functions* that are application node-specific. The ROS section of the interface includes structures necessary for Publish-Subscribe communication. These include *Incoming Queues* and *Outgoing Queues* that hold the (serialized) messages temporarily before processing.

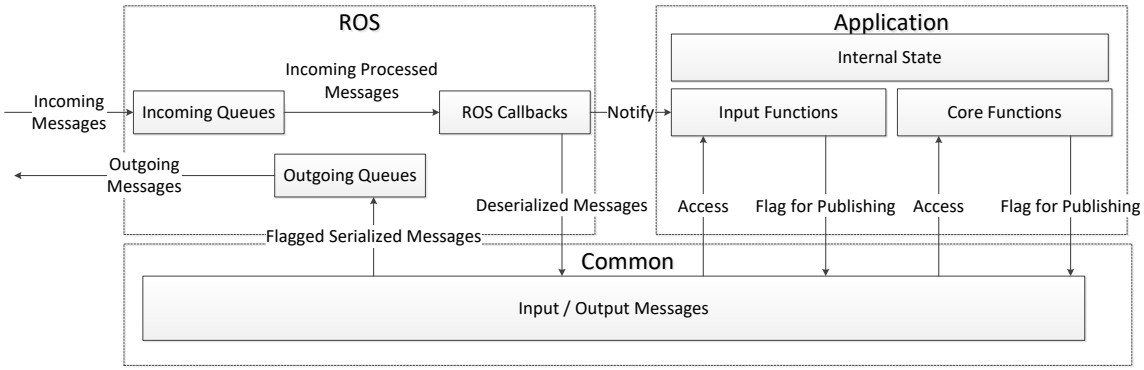

Fig. 22. ROS Interaction Detailed View.

In addition, ROS internally notifies Callback functions for a specific topic to inform when a message is processed from the Incoming Queues. The callbacks deserialize incoming messages and then call the appropriate Input function for notification. The Common section indicates the data that can be accessed from both sides of the interface. This includes the input / output messages that contain the data content (i.e. SerialObjects) that is to be communicated.

4.4.2 Build-Level

Each application node must be compiled into an executable program in ROS environment to work correctly with the communication layer. This includes linking to ROS's client libraries to access the ROS API functions. ROS's low-level build system, *catkin*, is able to compile programs and additional libraries that can be further linked in other application nodes/libraries.

Each ROS *package* contains source code, configuration files, and build/make files for identifying what code is to be included when compiling an application node. The ROS *packages* for the framework are divided into packages for nodes and packages for libraries. This is so that commonly used code, such as the base *Node* class or *SerialObject* classes, can be stored in a library that can be linked instead of copied with each application node. It also separates the definition of the base classes from the application node in a natural manner to facilitate replaceability.

To setup the system, the node topology of system can be defined in a configuration file known as a *launch file*. Nodes can also be executed manually, but *launch files* are chosen for ease-of-use and maintainability. Launch files are written in XML format. An example of a launch file is shown in Fig. 23. The file references two nodes, each providing attributes that specify the type of node to execute and a node identifier. The parameters under each node are used to specify the topic names made available for subscription /advertisement. An identifier is given to map to the topic name such that it can be referenced through the *FindTopicName()* function made available from the API.

```
<launch>
<node pkg="package1" type="package1_node" name="node1" output="screen"
launch-prefix="xterm -e " required="true">
  <param name="~input1" value="SUB_TOPIC_1" />
  <param name="~input2" value="SUB_TOPIC_2" />
  <param name="~output1" value="PUB_TOPIC" />
</node>
<node pkg="package2" type="package2_node" name="node2" output="screen"
launch-prefix="xterm -e " required="true">
  <param name="~input1" value="PUB_TOPIC" />
  <param name="~output1" value="SUB_TOPIC_1" />
  <param name="~output2" value="SUB_TOPIC_2" />
</node>
</launch>
```

Fig. 23. Example ROS Launch File.

The launch file is also used to execute the system nodes together on a single machine. The *roslaunch* tool is used to launching multiple ROS nodes locally and remotely via SSH [23]. If there are multiple machines, multiple launch files are used to specify the nodes that run on each machine.

An exception to this build environment is when working with nodes developed on Arduino. The programs (sketches) are developed and compiled using the Arduino IDE. A library called *rosserial* is linked to access ROS API functions through the Arduino [24]. The Arduino node then connects to the system via serial communication through a connected port to a *rosserial* node executed from a launch file. The Arduino implementation also requires slightly different implementation of the *Node* class library to build for Arduino. The differences are related to the different format of ROS messages in *rosserial* library compared to the regular ROS client libraries.

CHAPTER 5

FRAMEWORK DEMONSTRATION

In this chapter, use of the Test & Evaluation software framework is demonstrated through two use cases. In each, the use case is described, the structure and setup of the framework is illustrated, and details of the experiment results are provided. The use cases showcase testing of specific stages of the autonomous software. The first demonstration isolates the sense stage of a range finding sensor for use on a robot platform to illustrate the insertion of information across the virtuality-reality spectrum. The second demonstration isolates the planning stage of an autonomous rover performing obstacle avoidance to illustrate the use of information across the virtuality-reality spectrum.

5.1 Range Finding Demonstration

This use case involves demonstrating the Test & Evaluation framework for testing a range finding sensor in virtual, augmented, and physical reality. The range finding sensor is paired with a compass providing heading to plot cartesian coordinates of detected points in the environment. Physical sensors are mounted on a turntable, shown in Fig. 24, so they can rotate to sense the environment (translation is not supported as there is no location or motion sensor involved). The compass data is always produced as real, while the range finder data can be either real or virtual. This allows the system to detect objects in the virtual environment, physical environment, or augment the physical environment with virtual objects.

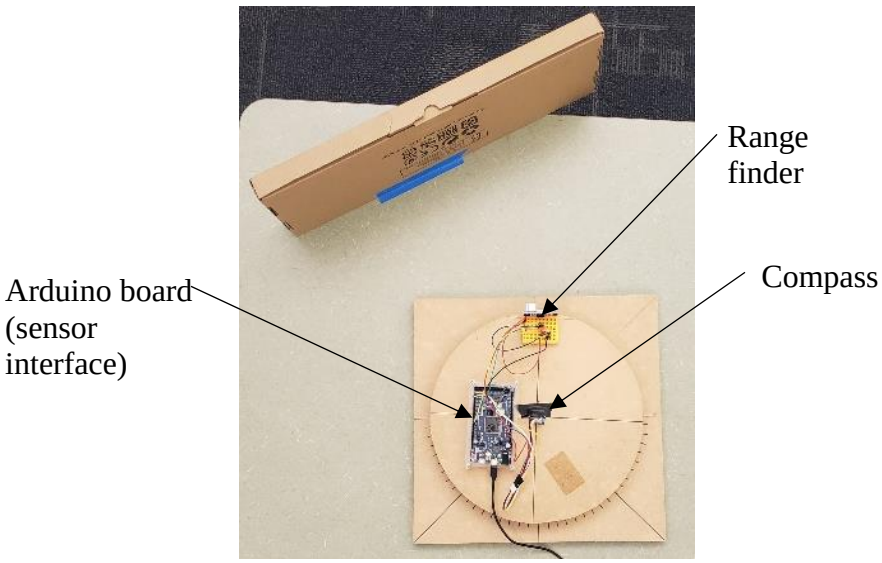

Fig. 24. Range Finder Hardware Setup.

Data from the physical range finder is provided as a distance in centimeters, and rotating the turntable allows distances from one or more objects to be detected and plotted. Data from a virtual world containing virtual objects involves defining the location and orientation, with location being FIXED and orientation being provided from the compass sensor to align the avatar representation of the platform with the real environment. Fig. 25 illustrates the coordinate system adopted in this example. Given location and orientation of the avatar, the distance to the nearest objects in the given heading are computed and returned as truth.

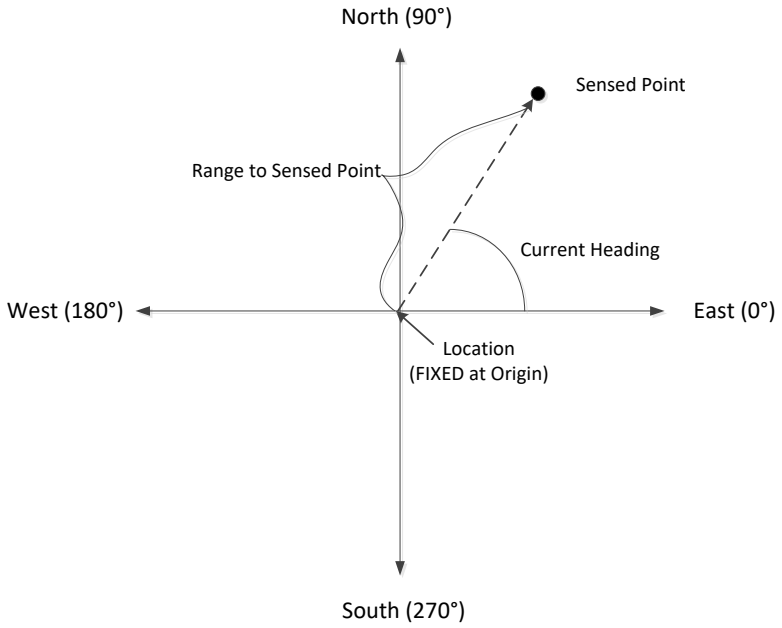

Fig. 25. Range Finder Coordinate System.

5.1.1 Setup

To start building this application, the major components of the system must be defined and mapped onto appropriate nodes in the framework as shown in Fig. 26. The setup is composed of a total of six nodes to include:

- Arduino Interface (Sensors) – Queries the compass and range sensor connected to the Arduino board and publishes the sensor data to the framework
- Custom Environment – Defines an environment for range detection comprised of simple geometric shapes
- Virtual Range Finder Model – Models error for the virtual range finding sensor
- Sense (Plot) – Provides visualization of sensor data as points on a graph
- Range Combiner – Performs selection and augmentation on physical and virtual range data
- Heading Single-Valued - Performs single-valued operation on physical heading data

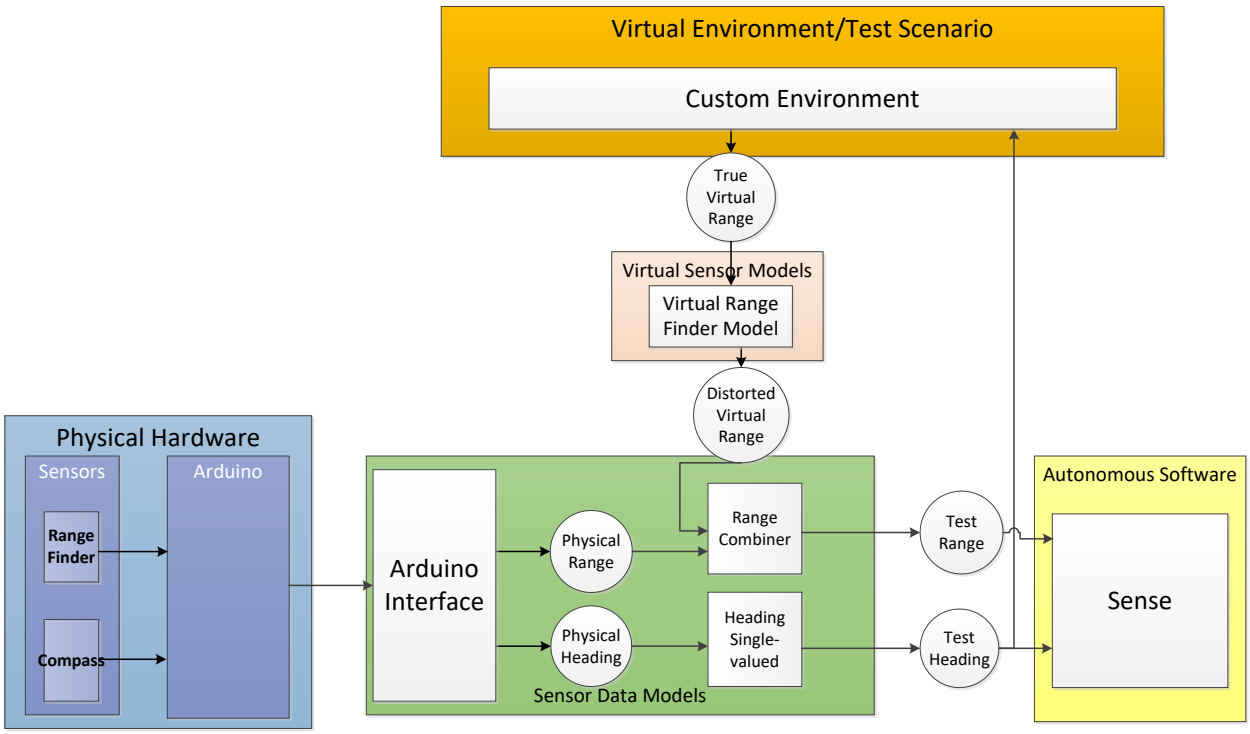

Fig. 26. Range Finder Framework Structure.

This section will now discuss each of the nodes to describe their purpose, behavior, and specific details to their construction.

5.1.1.1 Arduino Interface

The Arduino interface in this demo is used to obtain values from sensors to supply input to the framework. This node is implemented as an Arduino sketch which is uploaded to the Arduino board. The node advertises two topics for publishing: one for the physical heading and one for the physical range. The *Application* class features two initialization functions and two core functions to setup and query each sensor for data, respectively, and publish the data to the appropriate topic.

The hardware utilized includes an Arduino board, the range finder, and the compass sensor. An Arduino Due is the platform for connecting and running the physical sensors. A laptop computer supplies the Arduino with power and allows it to communicate through serial communication. The range finder used in this example is an HC-SR04 distance sensor that is able to detect obstructions up to 400cm. The compass sensor is an HMC5883L 3-Axis digital compass that measures heading offset from local magnetic field with 1 to 2-degree accuracy.

Fig. 27 illustrates how the HC-SR04 is connected to the Arduino board. The HC-SR04 requires a 5V voltage supply provided from the Arduino Due. The HC-SR04 works by sending a pulse 40 kHz ultrasonic wave and detecting whether a reflected pulse is received. The distance can then be calculated by measuring the time difference between sending and receiving the signal. The device has an input trigger signal for initiating each pulse wave and output echo signal for returning the time difference.

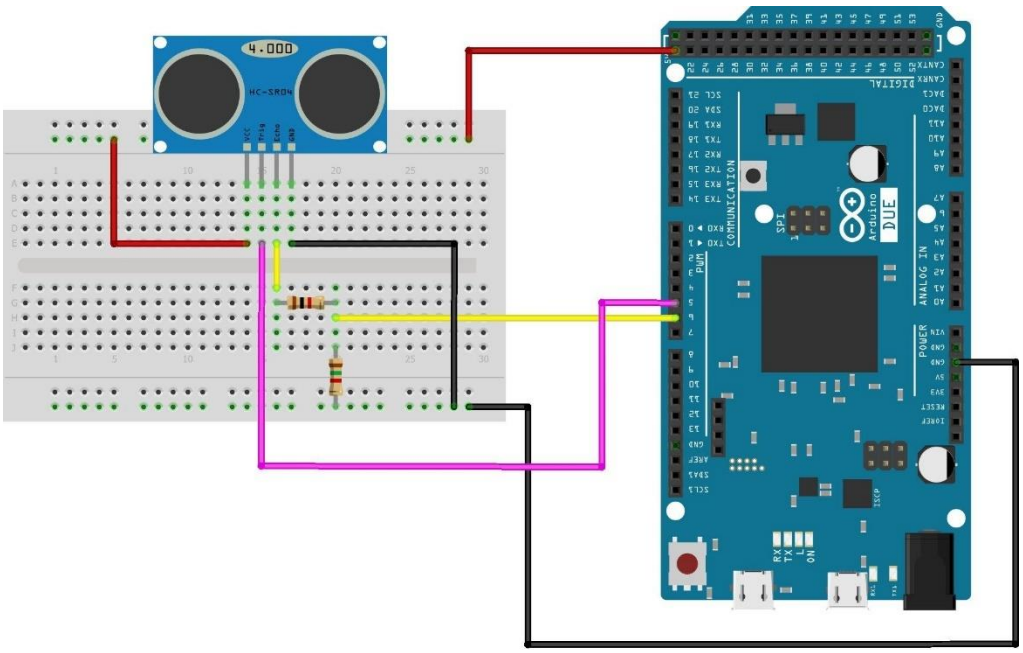

Fig. 27. HC-SR04 Hardware Setup.

The algorithm for triggering the pulse and computing distance is given in Fig. 28. The distance is then computed as half the time duration times the speed of sound (in cm/uS).

```
write LOW signal to trigger PIN to reset
delay for 2 microseconds
write HIGH signal to trigger PIN
delay for 10 microseconds
write LOW signal to trigger PIN to signal to start pulse
obtain time duration from echo PIN signal
distance = time duration / 2.0 * 0.0343
```

Fig. 28. HC-SR04 Loop Routine.

Additionally, the echo signal has a voltage range of 3-5V which is larger than what PWM pins on the Arduino Due can handle normally (max is 3.3V). A series of resistors are added to reduce the voltage to a manageable range.

Fig. 29 illustrates the HMC588L connected to the Arduino Due. This device also requires a 5V voltage supply from the Arduino Due.

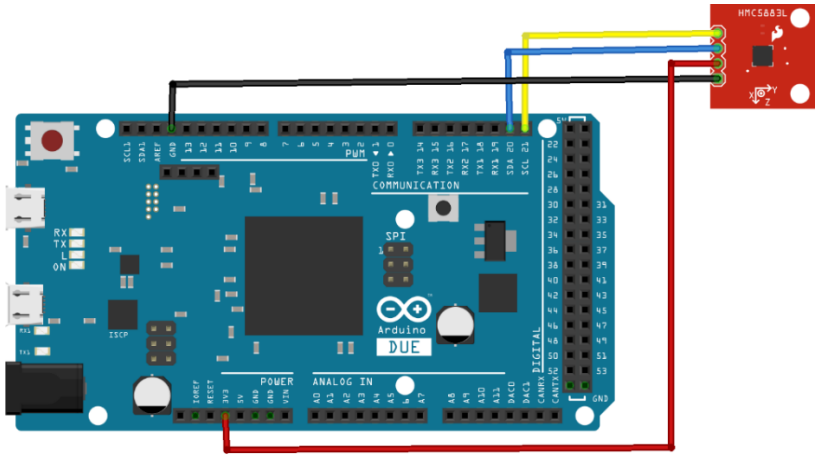

Fig. 29. HMC588L Hardware Setup.

The HMC588L works by measuring the orientation of the device's magnetic field offset from the local field in the area. While the device is able to obtain orientation in 3 axes, only the orientation around the vertical axis is published to the physical heading topic.

5.1.1.2 Custom Virtual Environment

The Custom Virtual Environment defines a world comprised of simple geometric shapes for virtual range detection. The geometric shapes are comprised of basic shape components such as line or curves or whole shapes such as ellipses. The virtual environment also maintains an avatar of the system in the virtual world. The node subscribes to the physical heading topic and publishes to a topic for true, or undistorted, virtual range. On the event of receiving new heading data, an intersection is computed between each of the shapes and a vector oriented along the current compass heading. The Euclidean distance can then be computed from the location of the avatar (in this case the location is FIXED at the coordinate system's origin) and the intersection point, the minimum distance of which is provided as true virtual range. The general routine for detection is shown in Fig. 30.

```
Initialize minimum distance to infinity
For each shape component in shapes:
        Compute intersection of heading vector with shape component
        If intersection exists
                Compute euler distance to intersection
                If distance < min distance
                        Set min distance to distance
Assign virtual range as minimum distance and flag for publishing
```

Fig. 30. Virtual Environment Detection Routine.

Each shape component is represented by a unique mathematical equation. The intersection is computed by solving for the coordinates ‘x’ and ‘y’ given a linear system of equations formed by the heading and the corresponding equation for the shape component. The equations are summarized in Table 3. Note ‘θ’ refers to the avatar’s heading.

Table 3. Details of shape components used in the Virtual Environment.

Shape Component	**System of Equations**	**Intersection**
Line	$Ax+By+C=0$ $-x\sin\theta+y\cos\theta=0$	$m=\frac{\sin\theta}{\cos\theta}$ $\begin{cases} x=-\dfrac{C}{A+mB},\ y=mx;\cos\theta\neq 0 \\ x=0,\qquad y=-\dfrac{C}{B};otherwise \end{cases}$
Quadratic	$Sx^2+Tx+Uy+V=0$ $-x\sin\theta+y\cos\theta=0$	$m=\frac{\sin\theta}{\cos\theta}\qquad D=\sqrt{(T+m)^2-4AC}$ $x=\dfrac{-B+D}{2A},\quad y=mx$
Ellipse	$\dfrac{(x-h)^2}{a^2}+\dfrac{(y-k)^2}{b^2}=1$ $-x\sin\theta+y\cos\theta=0$	$m=\dfrac{\sin\theta}{\cos\theta}\qquad \varepsilon=-k\qquad \delta=mh$ $x_{1,2}=\dfrac{hb^2-ma^2\varepsilon\pm\sqrt{a^2m^2+b^2-\delta^2-k^2+2\delta k}}{a^2m^2+b^2}$ $y_{1,2}=\dfrac{b^2\delta+ka^2m^2\pm abm\sqrt{a^2m^2+b^2-\delta^2-k^2+2\delta k}}{a^2m^2+b^2}$

Note the process may compute intersection points that are behind the avatar or outside the bounds of the shape component. To determine if an intersection point is valid, the intersection(s) is computed and then compared to boundary conditions of each shape component and the forward vector form by the sensor given the orientation of the avatar. Intersection points that are

facing away from the heading or are outside of the boundary of the shape are treated as if the intersection does not exist.

5.1.1.3 Virtual Sensor Model

As shown, while the Virtual Environment provides truth, it does not account for the irregularities and error associated with a physical sensor. The true virtual range represents the ideal distance, if sensor were perfect and provide the actual distance from a virtual obstruction. However, to accurately test the autonomous software, virtual distance data should emulate physical distance data. To achieve this, data was collected from the physical range-finder to capture sensed and real distances from various object shapes and materials. The data was used to build a model to modify truth data provided from the virtual environment.

Table 4 provides the results of input analysis done to estimate a parametric distribution for modeling sensor error in the virtual sensor model. The bolded entry represents the distribution that was chosen.

Table 4. Sensor Model Analysis Results.

Function	**Sq. Error**	**KS statistic**
Weibull	0.00421	0.145
Normal	**0.00473**	**0.12**
Gamma	0.011	0.142
Erlang	0.0166	0.146
Uniform	0.0187	0.125
Beta	0.0226	0.112
Triangular	0.0226	0.273
Lognormal	0.0253	0.168
Exponential	0.0718	0.229

The error distribution is estimated by taking the difference between actual and sensor distances over a number of actual distances from the sensor. Common probability density functions were tested against the empirical distribution from 16 difference data points using statistical tests in input analysis (e.g. chi-square and KS test).

Of the results, the parameters for a normal distribution were computed by maximum likelihood estimation for normal distribution. The normal distribution was chosen for its low mean square error to the data and its ability to account for values outside of those recorded. Fig. 31 shows the fitted distribution.

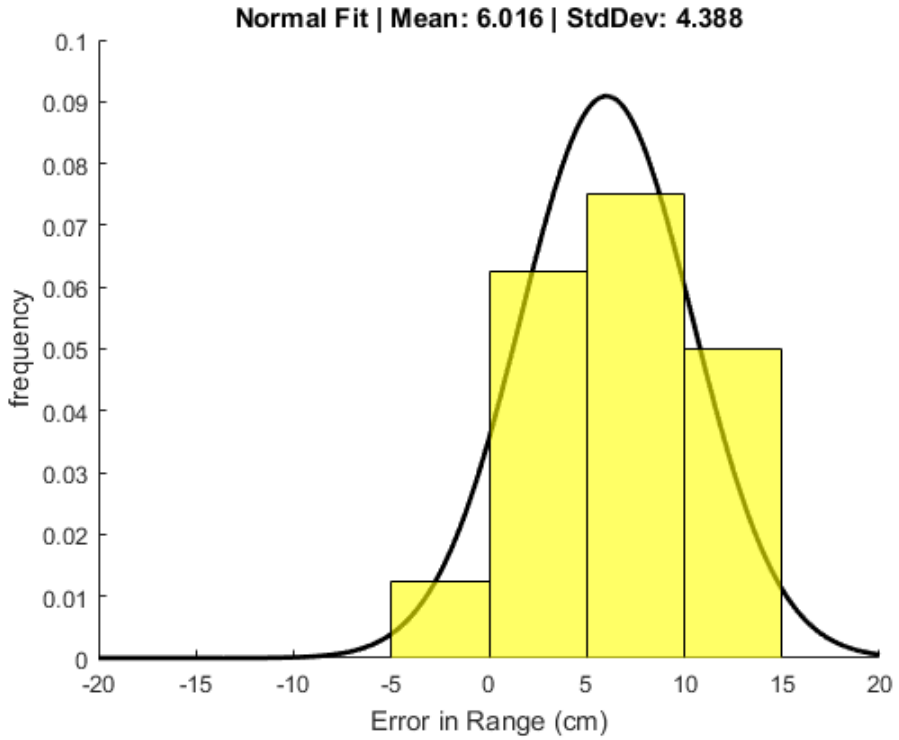

Fig. 31. Fitted Sensor Model Error Distribution.

Note, this only accounts for some of the error associated with the HC-SR04 and does not consider more complex issues associated with an ultrasound sensor such as reflection on

irregular or oriented surfaces. Ideally the resulting model would prevent the autonomous software from being able to detect the source of data by the form of the data. This would require more complete sensor modeling than done for demonstration purposes here. The process can be very complex, for instance the simple ultrasonic range finder employed may produce no data if the angle of the object being detected is too steep, or even worse, greatly chaotic results.

5.1.1.4 Range Combiner

The Range Combiner performs selection and augmentation on physical and virtual range data. Physical sensor data is obtained using an Arduino node and published to topics for the sensor data model. Virtual sensor data is obtained from the virtual environment and distorted through virtual range finder model. The data is then manipulated based on the following modes of operation:

- 0 (physical range finder data) – sensor data from the physical range finder is passed through.
- 1 (virtual range finder data) – sensor data from the virtual range finder is passed through.
- 2 (augmented range finder data) – sensor data from the physical range finder is augmented with data from the virtual range finder. This simple demonstration involves just taking the minimum of the two data. Other sensors would involve much more complicated augmentation (consider a camera image).

Based on the mode of operation, the Combiner waits until physical range, virtual range, or both are received before performing the selection or augmentation. Additionally, the Range Combiner has a minimum and maximum range limit. Input values that are outside the limits are passed on as infinity.

5.1.1.5 Heading Single-Valued

The heading single-valued model from the HMC558L. Note that this single-valued model is technically implemented as a *Combiner* class only without a virtual or augmented mode of operation. The *Heading Single-Valued* only has a single mode of operation:

- 0 (physical heading data) – sensor data from the physical compass is passed through.

The heading single-valued model also performs a conversion from the coordinate-system the HMC588L uses and the what is used internal to the autonomous software. Fig. 32 shows the conversion between the compass coordinate system and the coordinate system used by the autonomous software sense stage.

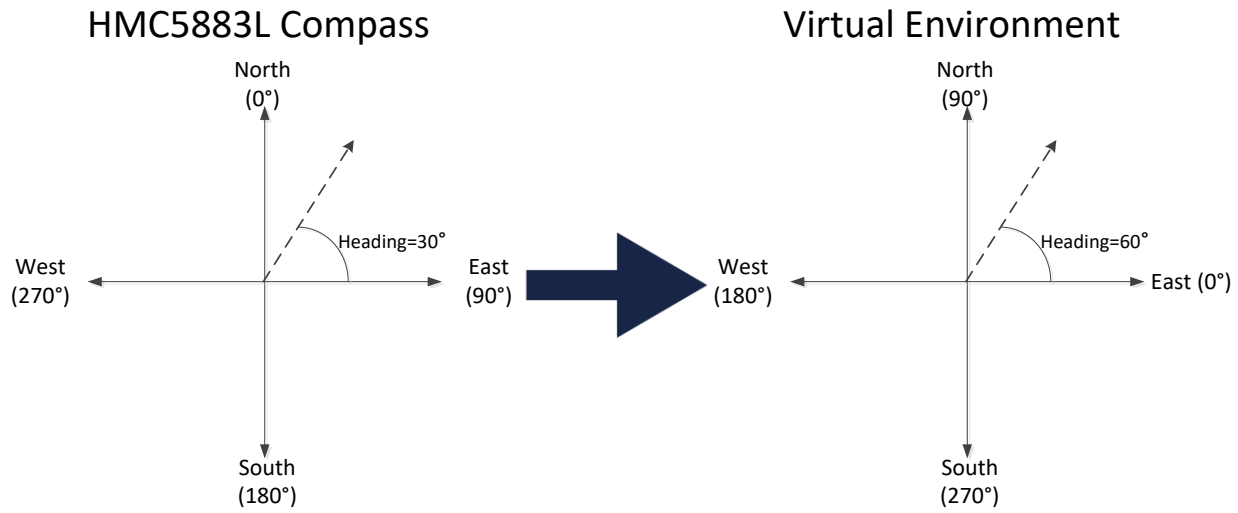

Fig. 32. Compass Coordinate Conversion.

5.1.1.6 Sense (Plot)

The Sense node acts as the autonomous software under test in this example. The node accepts sensor data from the framework and provides a basic visualization of the sensor data as points on a 2D cartesian graph. The data points are drawn in a dynamically as the sensor is

rotating and new heading and range data is received. The visualization is generated by utilizing the Qt graphics library.

Qt has its own procedures and events for handling window processing, drawing, and device input (such as from keyboard and mouse). Each Qt application must allow Qt API control of the program to perform its necessary activities. This creates the situation where there are two control loops within the Sense node as the framework API has its own control loop that must run regularly to process the sending and receiving of data on topics. As a consequence, the Qt portion must be run on a separate thread to prevent blocking the Publish-Subscribe processing. Fig. 33 provides an illustration of how both Sense and Qt both have their own independent control loops.

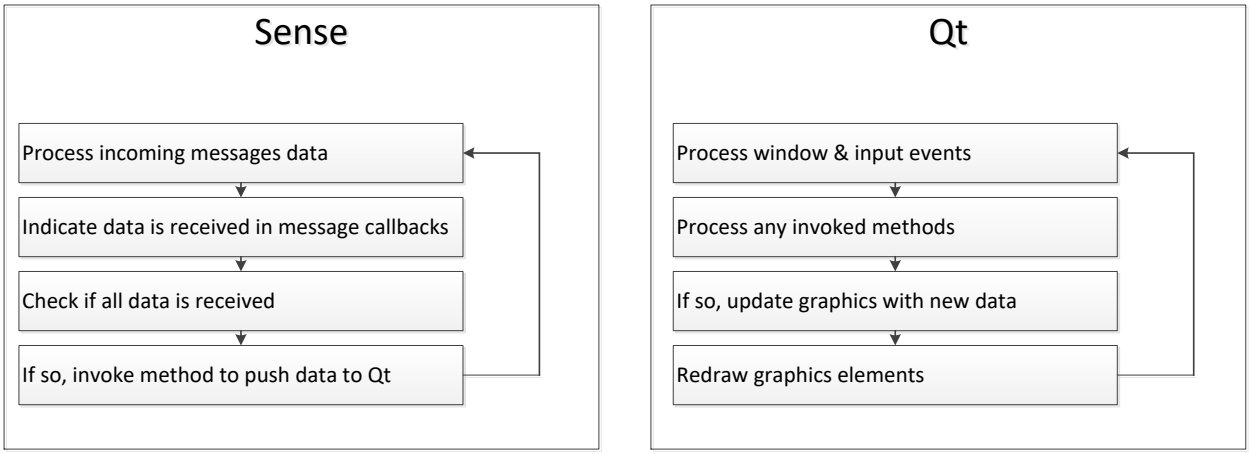

Fig. 33. Multithreaded Processing for Qt.

The main thread of the node is assigned to Sense. This thread processing events for receiving range and heading data and pushing the data to Qt once both are received. The other thread manages the graphics control loop that includes processing normal window or keyboard events,

processing data pushed from the main thread, updating the graphics with the new data, and then redrawing the graphics objects on the screen.

Fig. 34 shows a sequence diagram of the interaction between the two threads. Most of the activities are performed independently which allows both to handle timed updates (such as refreshing the screen) in their own manner. The primary interaction comes when data must be pushed to be available Qt thread to update the graphics objects. The Sense thread first processes incoming messages as per the regular *Node* control loop and indicates when each (range and heading) is received.

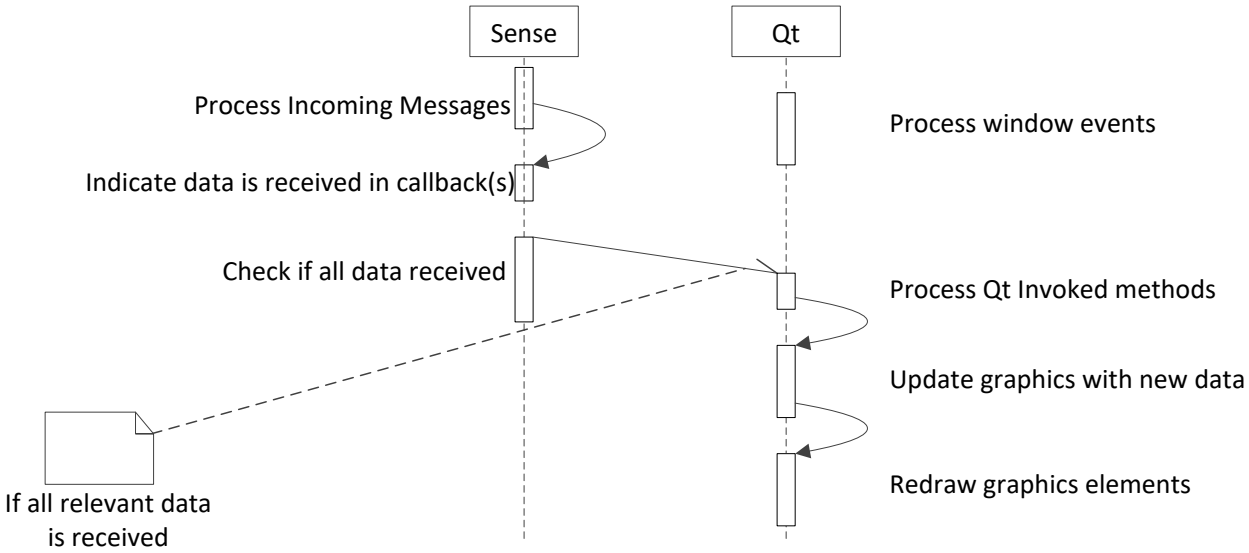

Fig. 34. Qt Thread Interaction.

Once both are received, the Sense thread makes an asynchronous call to a method in corresponding Qt class, providing the data to push as parameters. The Qt thread can then use the data to update graphics (such as adding point to the main plot). Regardless if a call was made, the Qt thread redraws the graphics elements and window to reflect the current state of the visualization.

5.1.2 Experiment

This section describes the experiment details undergone to demonstrate the range finder under different reality modes. This begins with a description of the environment scene in which the range finder will operate. The range finder is then set to sense the environment in each reality mode with results overlaid to view accuracy. This is followed by a discussion of the results of applying the framework to the range finder.

5.1.2.1 Range Finder Scene

In this experiment, the idea is to build a scene and vary the reality mode to demonstrate the effects of VR and AR on the Sense stage of the range finder. Fig. 35 shows the progression of the scene across virtual, augmented, and physical reality modes. In the scene, the robot is situated at the center of the environment and initially faces North. In the virtual scene, a simple rectangular shape is placed at some distance in front of the range finder (roughly 50-55 cm). As we move to augmented reality, a physical rectangular object is placed to the left of the sensor at a similar distance and different orientation. This allows the real and virtual objects to occlude each other partially. The third reality mode removes the virtual object from consideration.

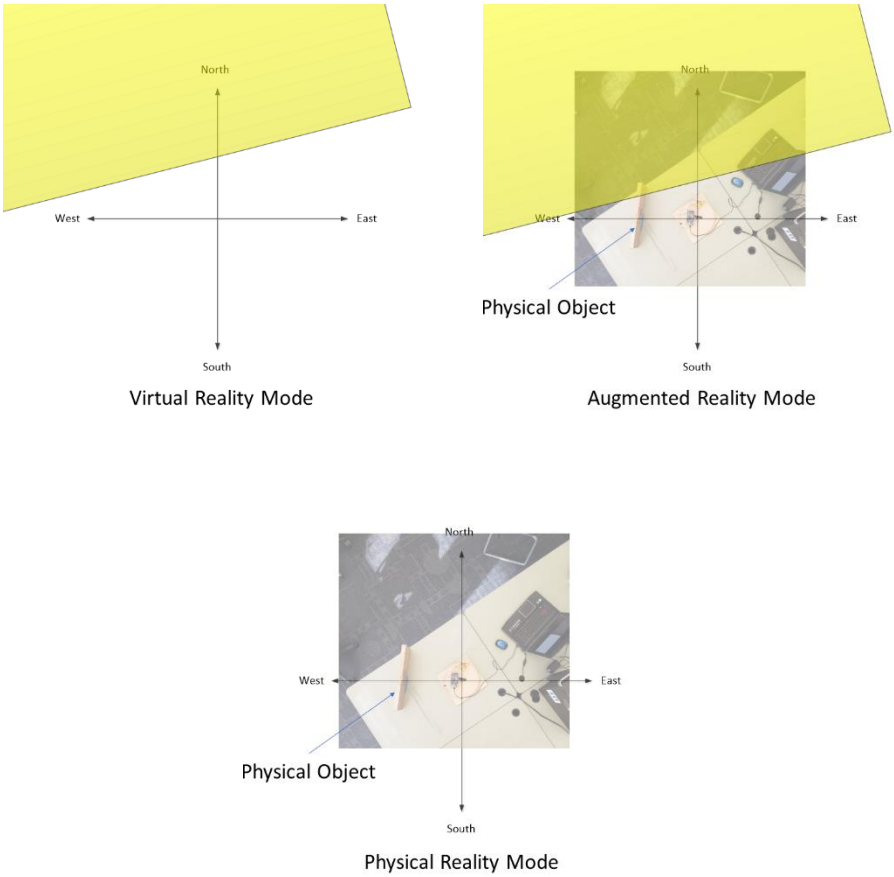

Fig. 35. Range Finder Scene in different Reality Modes.

5.1.2.1 Range Finder Results

By varying the reality mode of the Range Combiner, different environments are perceived without modification of the *Sense* stage. Fig. 36, Fig. 37, and Fig. 38 show plots of the resulting environment in the order of introduction of fidelity and movement across the virtuality-reality spectrum.

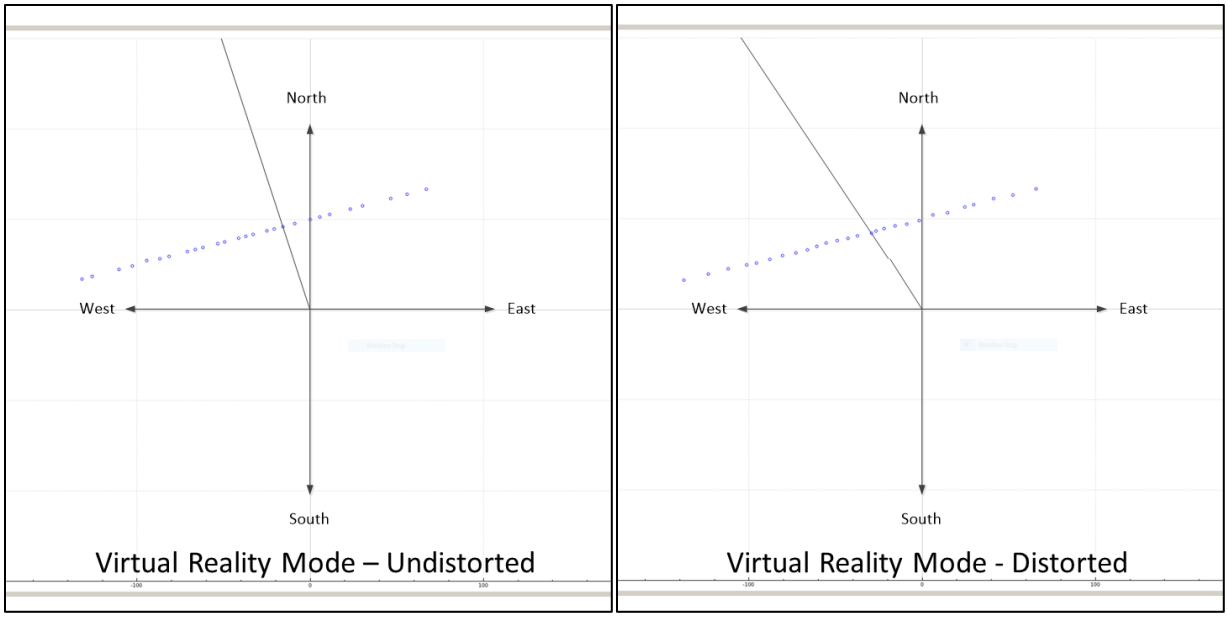

Fig. 36. Virtual Reality Plots.

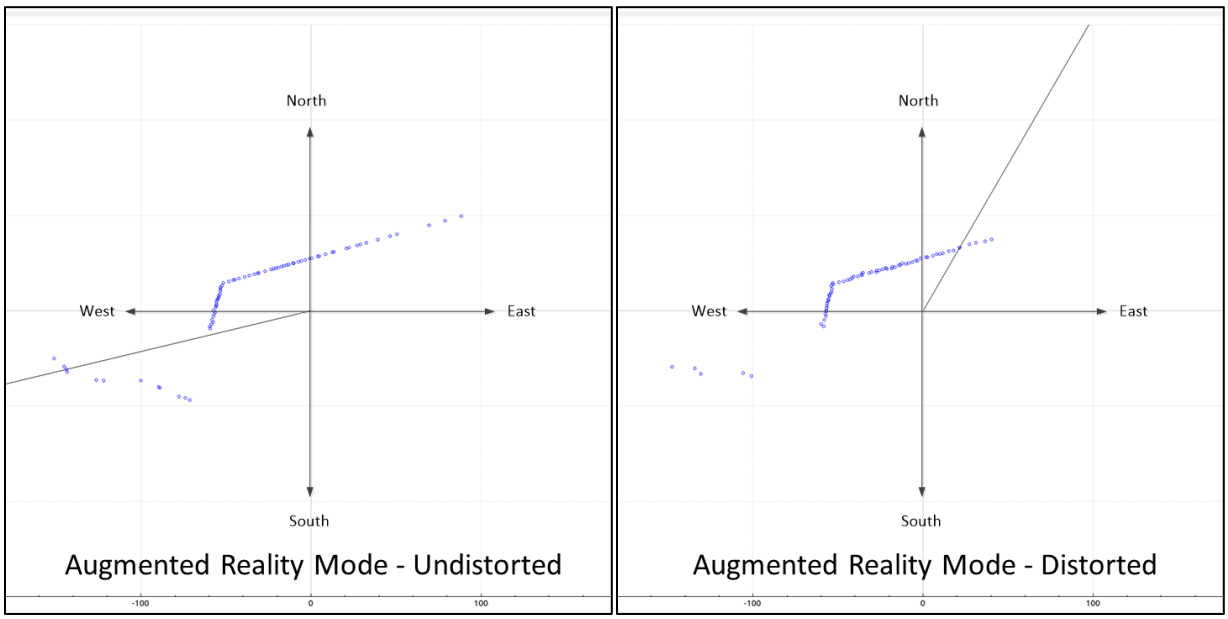

Fig. 37. Augmented Reality Plots.

Fig. 38. Physical Reality Plot.

5.1.2.2 Range Finder Conclusions

The results show that the range finder can be tested under different conditions in the virtuality-reality spectrum without modification of the autonomous software. First, the framework is able to isolate virtual data to test the *Sense* stage in a simulated environment. This illustrates that a basic environment can be perceived. Both physical and virtual range data can then be augmented, and the interaction can be modeled separately from the *Sense* stage of the autonomous software. Finally, the framework is also able to isolate and test with only physical data, showcasing that the fully integrated system can be tested with the same framework structure. Additionally, error models can be applied to the virtual data without having to modify the virtual environment or the autonomous software. The error is also modeled as a separate node and is, therefore, decoupled from any changes in the virtual environment. However, the framework also introduces issues in synchronization and a time lag between the *Sensors* and *Sense* stage. This can be seen notably in the virtual and augmented reality mode, where the lag

causes (small) distortions in the virtual data that would otherwise not exist. The lag is likely due to a different rate of computation between virtual data (produced by the virtual environment) and physical data (published from the Arduino interface). Further research may work to adjust the framework to accommodate the synchronization issues.

5.2 Obstacle Avoidance Demonstration

This use case demonstrates the Test & Evaluation framework for an autonomous rover avoiding obstacles and boundaries encountered in the environment. The example focuses on the use of information in the virtuality-reality spectrum. Unlike the range finder use case, this example does not use any physical sensor devices. The autonomous software is entirely operating in a virtual environment. An autonomous planning stage directs the rover's actuators (i.e. wheels) when obstacles are detected. The actuator data can be used to control a virtual avatar of the rover in the virtual environment. The data can also be used to control physical actuators on a real-life rover, shown in Fig. 39.

Fig. 39. Physical Rover Chassis.

The virtual environment is composed of three coordinate frames: a global coordinate system, a vehicle-carried coordinate frame, and a local coordinate frame. This is similar to the Local NED (North-East-Down) coordinate system [25]. Fig. 40 provides an illustration. The global coordinate system is a two-dimensional space measured in centimeters along a horizontal and vertical axis. For this demonstration, the origin of this global coordinate system and the orientation of its axes are mapped arbitrarily to the area used at the time of testing. The vehicle-carried coordinate system is centered on the position of the entity within the global coordinate system.

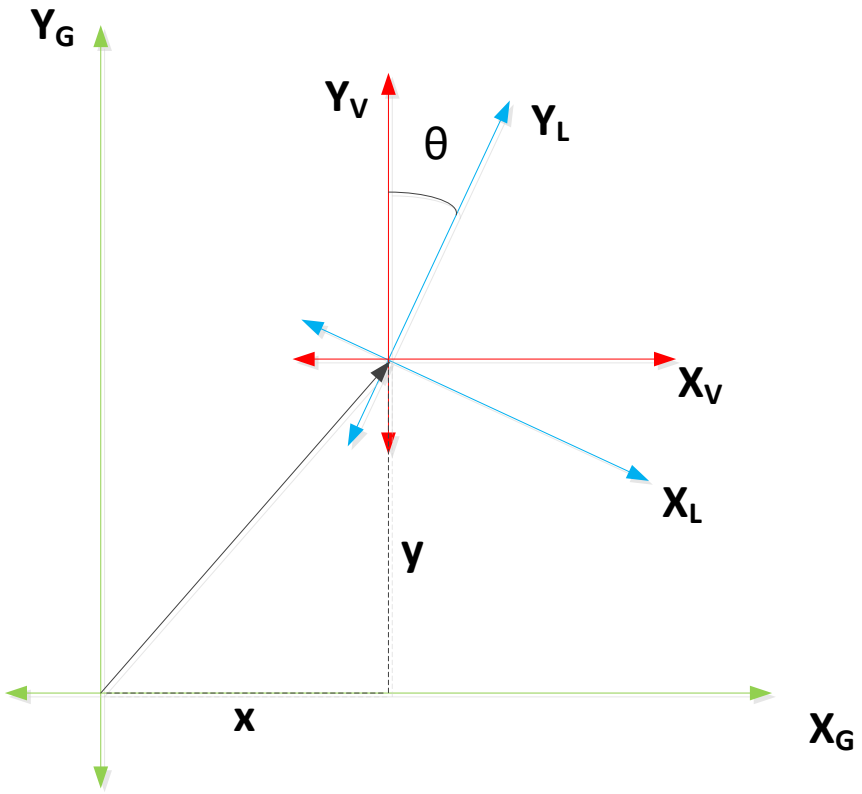

Fig. 40. Coordinate Systems Utilized in Obstacle Avoidance.

The axes of the vehicle-carried coordinate frame are oriented in the same direction as the axes of the global coordinate frame. Finally, the local (or body) coordinate frame is centered on the position of the entity but with axes that are aligned with the orientation of the entity relative to the vehicle-carried coordinate frame.

As such, the coordinates of each entity within the environment can be described with three parameters: {x,y,θ}, where 'x' is the distance in cm from the global to the vehicle-carried coordinate frame's origin along the horizontal axes, 'y' is the distance in cm from the global to the vehicle-carried coordinate frame's origin along the vertical axes, and 'θ' is the orientation of the local coordinate frame in degrees relative to the vehicle-carried coordinate frame.

5.2.1 Setup

Similar to the range finding example, the major components of the system are defined and mapped onto the appropriate nodes within the framework as shown in Fig. 41. Some of the components have similar responsibilities as the ones in the range finder with a few notable exceptions. The major components of this demonstration are provided as follows:

- Arduino Interface (Actuators) – Sends signals to wheel actuators based on actuator control data received from the framework
- Custom Environment – Defines the environment comprised of boundaries and obstacles and an avatar of the rover
- Virtual Rover Model – Computes motion (translation and rotation) of virtual rover over time based on current wheel actuator data
- Planner – Determines wheel actions for autonomous rover based on detected objects and current location and orientation

- Detection Model – Performs selection on obstacles detected in environment
- Rover Pose Model – Performs selection on the rover's position and orientation data made available to the autonomous software
- Wheel Splitter – Relays wheel actuator data to both physical and virtual actuators
- Visualization – Presents a visualization of the virtual environment and current rover position and orientation

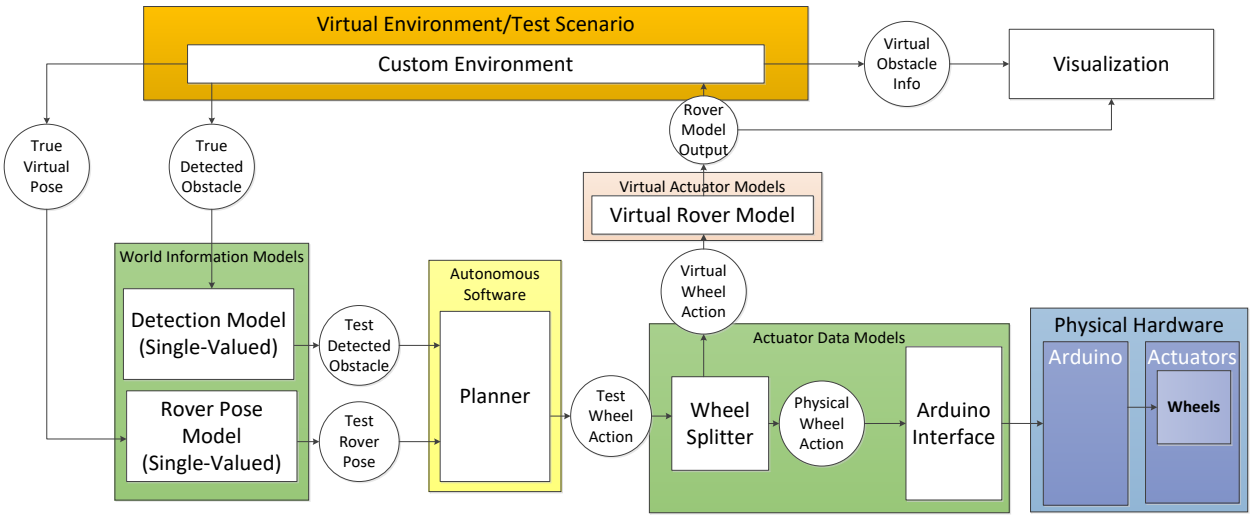

Fig. 41. Obstacle Avoidance Framework Structure.

This section will discuss each of the nodes for obstacle avoidance including their purpose, behavior, modeling details, and related information.

5.2.1.1 Arduino Interface

The Arduino interface is used to communicate with the physical rover's wheels. This node is implemented as an Arduino sketch uploaded to the Arduino board. The sketch subscribes to one topic to receive the physical actuator control data. The class also features an

initialization function and core function to setup and send signals to the wheel actuators, respectively.

The hardware utilized include an Arduino board, a Sabertooth dual 25A motor controller, and four IG52-04 motors (the two rear motors include encoders), and a rover chassis. The IG54-04 motors are powered by a 24V battery regulated through the motor controller. The motor controller drives each set of motors (left/right) independently. The Arduino communicates with the motor controller through 2 PWM signals (one for each set of motors). The signal is an integer value from 1 to 255 whose values are partitioned into two ranges where each range sets the velocity of one set of motors. These values are further mapped to the range [-1,1] to simplify control, allowing each set of wheels to either go full forward, full reverse, or stop. Table 5 shows the range of values and their corresponding motor set and velocity.

Table 5. Signal values used for controlling Wheel Motors.

Full Reverse	Stop	Full Forward	Full Reverse	Stop	Full Forward
1	63	127	128	191	255
-1	0	1	-1	0	1
Right Motors			Left Motors		

5.2.1.2 Custom Virtual Environment

The virtual environment in this demonstration is comprised of a set of obstacles and a boundary around the environment. It also maintains a virtual avatar of the rover within the environment, allowing the environment to provide information relative to the rover position. The obstacles are represented as shapeless points within the environment. Each obstacle is

described by a number identifier and a position within the environment. The boundary sections are represented and described as line segments. The environment node listens for pose information to update the position and orientation of virtual rover avatar. The node also performs collision detection and publishes information about detection events that occur during the computation.

A vehicle avatar is maintained in the environment and is described by the parameters as shown in Table 6. This includes parameters for describing the position and orientation of the rover within the virtual environment. It also includes parameters for defining the vehicle's ability to sense the environment. The node subscribes to the virtual location topic and publishes the event on which the avatar detects an obstacle or boundary.

Table 6. Virtual Rover Avatar Parameters.

Avatar Parameter	Description	Units
x	Distance of rover from global origin along the horizontal axis	Cm
y	Distance of rover from global origin along the vertical axis	Cm
θ	Angular displacement of rover around the local origin	Degrees
δ	Maximum distance that obstacles can be detected	Cm
ϕ	Half the field of view that obstacles can be detected within	Degrees

The virtual environment node registers an input function for notification of new pose data for the rover. On the event of receiving the data, the environment computes a projection with the available obstacles and boundary segments. Fig. 42 illustrates the detection of obstacles within the virtual environment. For each obstacle, a projection vector is computed and compared to the maximum detection distance and the field of view to determine if the obstacle can be detected. If

there are multiple obstacles are detected, the virtual environment keeps the obstacle that is closest to the rover as truth.

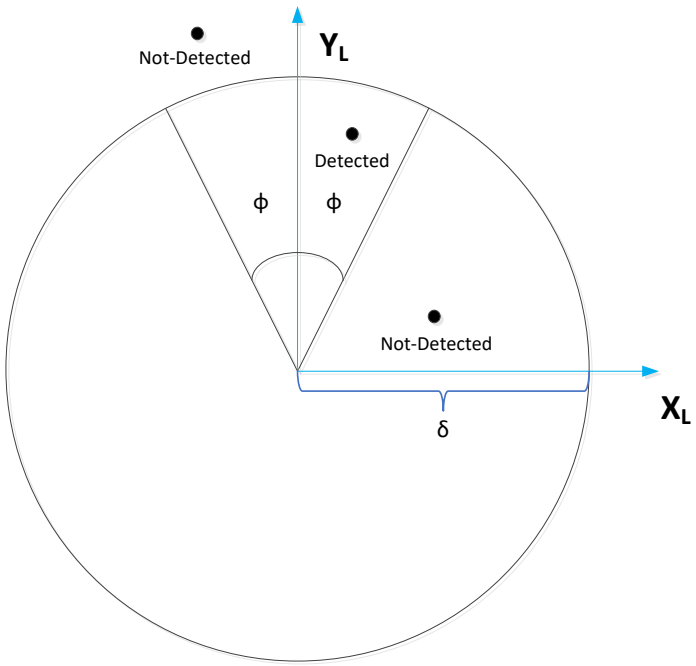

Fig. 42. Detection of Points for Obstacle Avoidance.

Fig. 43 illustrates the detection of boundaries within the virtual environment. The boundary is described using a line segment instead of a point. For each boundary segment, an intersection is computed between the segment and rays oriented by angle ϕ relative to the rover's local coordinate system. Intersections that are not in front of the rover or are outside the maximum distance threshold are disregarded. In the case of multiple (two) detections on more than one boundary segment, the environment assumes that the boundaries will intersect at a corner. The coordinates of the detection point are then estimated as coordinates from the two detected points with the greatest absolute value in the global coordinate system.

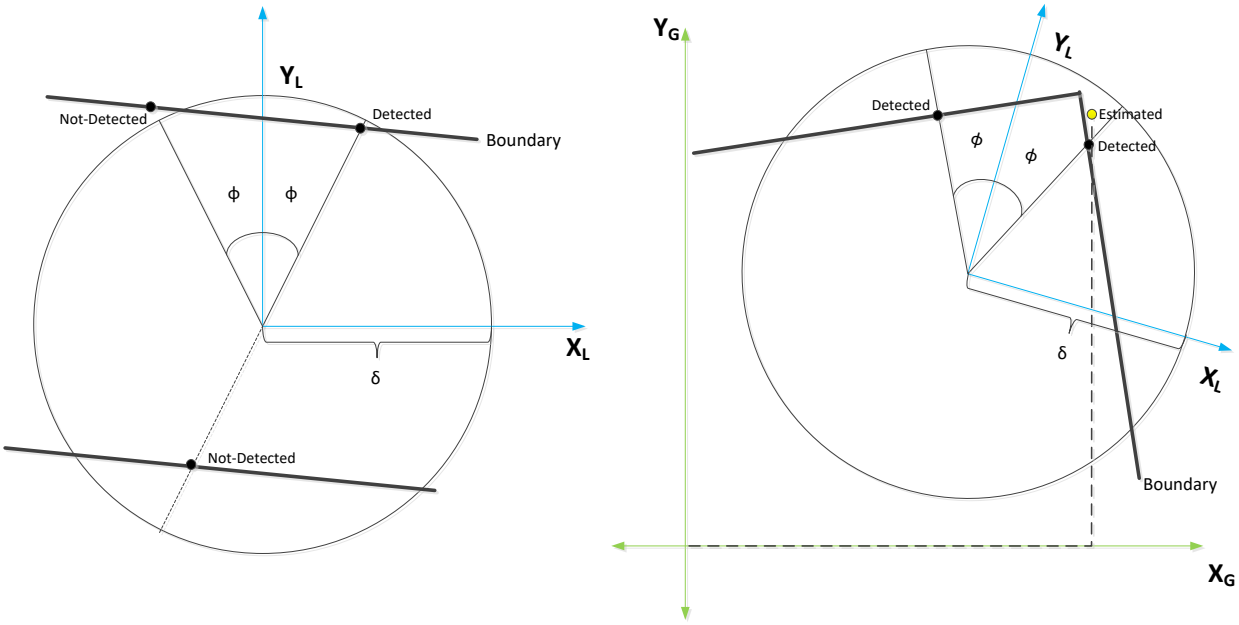

Fig. 43. Detection of Boundaries for Obstacle Avoidance.

After computing the closest detection, the environment publishes the event including the ID of the obstacle, the position of the obstacle, and the computed distance to the obstacle from the rover. In case of detecting a boundary or corner, the ID of the obstacle is not defined.

5.2.1.3 Virtual Rover Model

The virtual rover model is responsible for simulating the motion of the virtual rover over time to provide position/orientation to the virtual environment in the absence of localization hardware. It acts as a virtual counterpart to the Arduino interface/physical rover. The node accepts the actuator data to control the virtual rover's motion. The results of the model computation are a new pose for the virtual rover that is published to the framework.

The model is based on a kinematic model with three degrees of freedom. The model is simplified and is mainly used for proof of concept rather than physical accuracy. The parameters for the model include rotational speed (c_{Rot}), translational speed (c_{Trans}), and an axis of rotation

defined in two dimensions (a_L). Note that translational speed is in the local coordinate system. The model also depends on the current position and orientation of the rover (given by the tuple $\langle p_G \quad \theta_L \rangle$) and current state of the left and right wheels (w_0 and w_1, respectively). The rover state definitions are given in Equation 1. Note the subscripts of 'L' and 'G' are used to signify the local and global coordinate space, respectively, for each variable.

$$p_G = \begin{pmatrix} x_G \\ y_G \end{pmatrix} \qquad R_L = \begin{pmatrix} \cos\theta_L & -\sin\theta_L \\ \sin\theta_L & \cos\theta_L \end{pmatrix} \qquad m_L = \begin{pmatrix} \cos\theta_L \\ \sin\theta_L \end{pmatrix} \tag{1}$$

The computations for the model are divided into two parts: rotation and translation. Rotation may affect both the position and orientation of the rover, whereas translation only affects the position. The rotations effect on position depends on the current wheel state which, in turn, affects the axis of rotation. Equation 2 illustrates the intermediate formulas that are used in computing rotation and translation.

$$d_{Rot} = \begin{cases} 1, & w_0 < w_1 \\ -1, & w_0 > w_1 \\ 0, & otherwise \end{cases}$$

$$a_L = \begin{cases} \begin{pmatrix} x_L \\ 0 \end{pmatrix}, & w_0 < w_1 \\ \begin{pmatrix} -x_L \\ 0 \end{pmatrix}, & w_0 > w_1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, & otherwise \end{cases} \qquad d_{Trans} = \begin{cases} 1, & w_0 > 0, w_1 > 0 \\ -1, & w_0 < 0, w_1 < 0 \\ 0, & otherwise \end{cases} \tag{2}$$

$$\Delta p = c_{Trans} \cdot m_L \cdot \Delta t \cdot d$$

$$\Delta\theta = c_{Rot} \cdot \Delta t \cdot d$$

$$\Delta R_L = \begin{pmatrix} \cos\Delta\theta & -\sin\Delta\theta \\ \sin\Delta\theta & \cos\Delta\theta \end{pmatrix}$$

In both cases, a direction variable is used to determine a forward/CW or backward/CCW direction to apply the rotation or translation. The rotation operation then calculates an axis of

rotation to correct for off-balance wheels. Finally, a delta is calculated for rotation and translation based on the speed parameter for each and a time step to advance.

After computing the deltas for rotation and translation, the new pose can be computed as shown in Equation 3. The formula for p'_{Rot} transforms the position to center around the axis of rotation before applying the rotation. p'_G is then computed by applying the translation delta. The new pose is represented by the tuple $\langle p'_G \quad \theta'_L \rangle$.

$$\begin{aligned} p'_{Rot} &= (\Delta R_L)(R_L a_L + p_G) - (R_L a_L + p_G) \\ \theta'_L &= \theta_L + \Delta\theta \\ p'_G &= p'_{Rot} + \Delta p \end{aligned} \tag{3}$$

It is assumed that the rover model accepts the same type of data as the Arduino interface. This includes integer values for the left and right wheel within the range [-1,1]. It is also assumed that the axis of rotation only varies along the local X_L axis of the rover. This simplified the calculations. The experiment section (5.2.2) further states that weights are added to the physical rover to bring its axis of rotation into a similar alignment. Another assumption is that the translation speed and rotation speed parameters are constant given a specific wheel state.

5.2.1.4 Planner

The Planner node acts as the plan stage of the autonomous software in this application. The main goal of the planner in this application is to direct the rover to avoid obstacles or boundaries within the environment. This involves assessing the current state of the world representation and deciding to change the course of the rover or keep it the same. The change is then communicated by publishing new wheel actuator data.

The world representation for the planner is composed of the most recent detection within the environment and the rover's most recent pose within the environment. Both types of data are

used to determine what action the rover should perform to avoid collision. In addition, an internal timer to keep a particular action for a certain amount of time.

The actions of the planner can be summarized by the state machine shown in Fig. 44. The state machine is composed of three main states that correspond to the action the Planner directs the rover to make when that state is active. The state highlighted green ("Go Forward") is the initial state of the Planner. The planner stays in this state until a detection is observed.

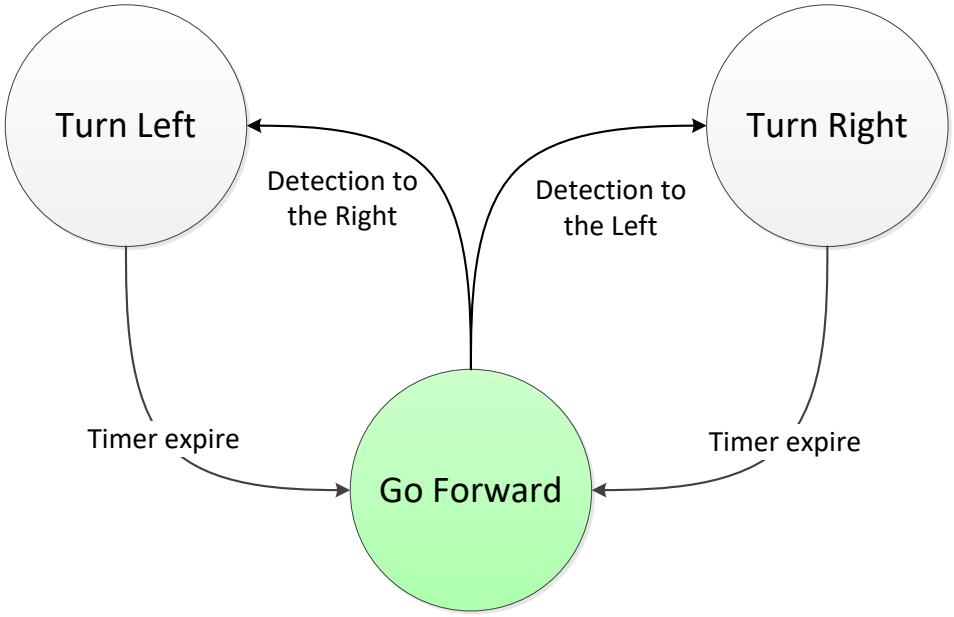

Fig. 44. Planner State Machine.

The Planner then determines whether the detection is to the rover's left or to its right based on the rover's current pose (position and orientation). If it is to the left, the Planner transitions to the "turn right" state to attempt to avoid the obstacle; if it is to the right, the Planner transitions to the "turn left" state. In either transition, the timer is set to a certain time in the future. The Planner's core function then iteratively compares elapsed computer time with the set time.

When the timer expires, the planner transitions back to the “Go Forward” state and the process can repeat.

5.2.1.5 Detection Model, Rover Pose Model, and Wheel Splitter

The detection model is a node for performing selection and augmentation of detection events within the framework. For the current demonstration, it is assumed that there are no physical detections and that all detections are from the virtual environment (either of an obstacle or boundary) from the virtual rover. Therefore, the detection model is configured as a single-valued node that passes through the data coming from the virtual side of framework. The model is still implemented as a *Combiner* in the anticipation of detections obtained from a *Sense* node. That is, the detection model acts as a placeholder to eventually be extended to a *Combiner* in the future.

Similarly, the Rover Pose model performs selection and augmentation of the position and orientation data required by the Planner. The node is also configured as a single-valued node that simply passes the virtual pose data from the Virtual Environment as no physical sensors are used for localization. At this time, the data does not require any conversion from the coordinate system used by the Virtual Environment as the Planner assumes the same representation. However, the model is available for potential conversions if nodes should be replaced.

The wheel splitter is a framework node that relays wheel actuator data to both physical and virtual actuators. The node also does not need to apply conversions to the data to work with the physical and virtual actuators. This is node is still made available to maintain isolation and allow for potential conversions of wheel actuator data should nodes be replaced.

5.2.1.6 Visualization

The visualization in this example is a separate node that observes data from the virtual rover and virtual environment and presents a visual depiction of the current state of both over time. The node observes information about the rover pose and obstacles within the virtual environment to construct a map of the virtual world. The visualization is also generated by utilizing the Qt graphics library.

Like the Sense (Plot) node in the range finding application, this node maintains two threads for regular communication and graphics updates, respectively. The sequence diagram is mostly the same with a difference in the type of graphics that need to be updated in the visualization; and, therefore, the type of data that needs to be communication between threads. Fig. 45 shows the graphics components of the visualization. This includes a graphic for the virtual rover avatar and graphical representations for each obstacle and boundary within the environment.

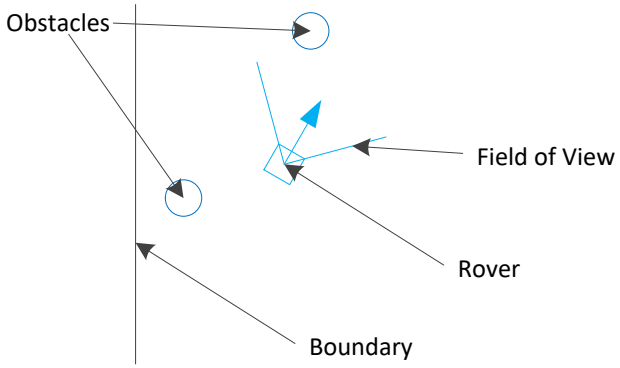

Fig. 45. Visualization of Virtual Avatar and Environment.

Note that the size of the obstacles within the visualization are not indicative of actual size. The size is only made as such to make it easier to view. Additionally, the visualization shows the rover pose and obstacle positions from the global coordinate frame.

5.2.2 Experiment

In this section, the experiment for testing obstacle avoidance will be described. This begins with a description of the scene the rover will perceive. This is followed by a discussion of the results of applying the framework to obstacle avoidance.

5.2.2.1 Obstacle Avoidance Scene

The experiment scene for this demonstration is set up in an enclosed virtual environment with obstacles that are scattered randomly within the room to provide variety. The idea is to mimic the environment that a Roomba would traverse in order to perform an operation such as cleaning or floor mapping [26]. As such, four boundary segments are defined at a distance of 100cm along each axis from the origin of the global coordinate frame as shown in Fig. 46. Additionally, 10 obstacles are created and placed randomly within 60cm radius of the global origin.

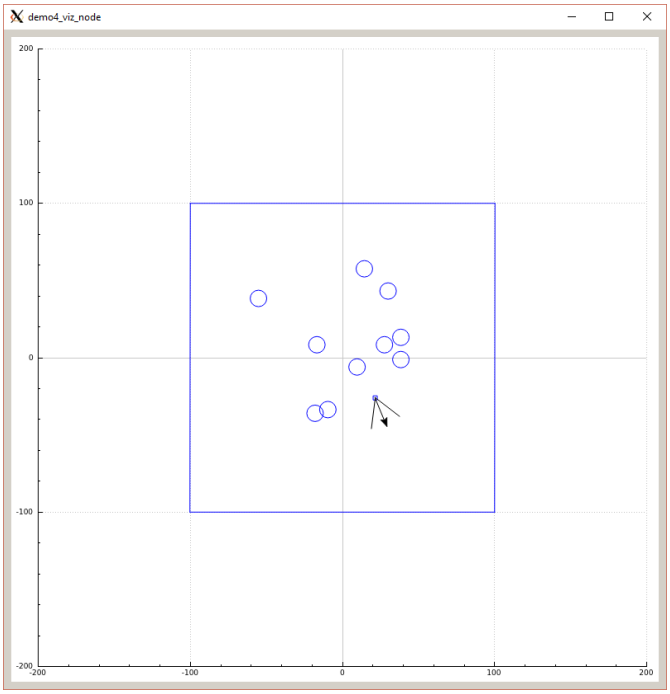

Fig. 46. Experiment Scene for Obstacle Avoidance.

Several parameters for sensing and motion are also set for the virtual rover avatar. This includes the maximum range and viewing angle (field of view) to detect any obstacle or boundary. It also includes the initial pose of the rover and parameters such as translational and rotational speed. These are summarized in Table 7.

Table 7. Experiment Parameters for Obstacle Avoidance.

Parameter	Initial Value	Units
X	0.0	Cm
Y	0.0	Cm
Θ	0.0	Degrees
Δ	20.0	Cm
Φ	30.0	Degrees
C_{Rot}	23.0	Degrees/sec
C_{Trans}	7.0	Cm/sec
x_L	5.0	Cm

5.2.2.2 Obstacle Avoidance Conclusions

This demonstration indicates promise in utilizing the framework to facilitate testing between simulated and physical systems. It highlights the capability of the framework to isolate the autonomous software from the system it controls. As such, the Plan stage is able to be tested controlling a simulated and physical system without modifications to the autonomous software. Additionally, the autonomous software Plan stage is isolated from the virtual environment. As such, the virtual environment can be replaced with an environment with greater fidelity or more resolution. This is also true for the virtual rover model. While the model localizes the autonomous system within the virtual environment, the model is a crude approximation of the rover's motion with assumptions that do not account for physical effects such as friction. In addition, other localization methods could be employed such as utilizing GPS sensing or the motor encoder data that could more accurately position and orient the rover within the virtual environment; although, these would only be relevant in an augmented or physical reality mode. As the virtual and physical system is decoupled from the autonomous software, modifications to add a higher fidelity model or an appropriate substitute would not impact the autonomous software stages.

CHAPTER 6

CONCLUSION

The Test & Evaluation process of autonomous software presents many challenges in handling the transition between early simulated systems and the fully integrated physical system. The framework presented in this thesis provides a promising architecture for supporting the testing of autonomous software over the course of its development. The framework isolates the autonomous software from the components of the testing environment (e.g. virtual environment), allowing the software to be tested provided the components can provide sufficient data required by the autonomous software. A Publish-Subscribe communication pattern is leveraged to support communication in the decoupled system. The framework test harness includes nodes (e.g. Combiner, Splitter, and Single-valued) to manipulate and route information as necessary to conduct different testing scenarios. The two use cases provide a proof of concept of the architecture and illustrate the framework is capable of supporting testing of stages of the autonomous software in mixed reality without additional effort to reconfigure for different environments, such as creating separate testing harnesses.

Further applications can now be explored. A development process may be defined to address parallel development across different development roles of the autonomous system. Several roles have been identified in this thesis; but a process is required to ensure external sources of information (i.e. virtual environment, virtual sensor models) are ready to allow testing of the autonomous software. Additionally, the use cases have only tested a single autonomous system in the environment at a time. Leveraging the framework, collaborative autonomous systems could be developed in both physical and virtual reality by isolating knowledge of

whether a collaborating system is, in fact, physical or simulated. For example, the Obstacle Avoidance use case has recently been expanded to include a virtual rover autonomous vehicle that can interact with the physical rover within the virtual environment as if it were another obstacle to avoid. The framework API could also be extended to include direct channel communication or command and control (C2) communication between collaborating systems in addition to the regular data communication via topics. There is also potential to develop human interfaces to visually observe the virtual and augmented environments while testing scenarios within the virtuality-reality spectrum. Visualizations can leverage the framework to isolate itself from the autonomous software or virtual environment.

While the system is now a valid proof of concept, there are still areas of potential improvement. The demonstration indicates issues with Publish-Subscribe communication that may need addressing. Time lag between certain nodes may be fine for low-risk systems but introduce problems with autonomous systems that require a high-level of accuracy. Additionally, the security requirements of each node must be considered. Should nodes assume the data they are provided is correct? This can lead to a question of integrity within the system, which can be a critical concern for autonomous systems that make decisions based off of the available data. ROS is known to have security issues [27]. Additionally, the amount of data communicated between nodes in the use cases is relatively minute. Research should keep in mind how increasing data requirements (such as the introduction of an 2D images) will impact the communication.

REFERENCES

[1] Goldman Sachs Research, "Drones: Reporting for Duty," [Online]. Available: https://www.goldmansachs.com/insights/technology-driving-innovation/drones/. [Accessed 1 February 2018].

[2] CAPITOL Technology University, "Masters of Science (MS) in Unmanned and Autonomous Systems Policy and Risk Management," [Online]. Available: https://www.captechu.edu/degrees-and-programs/masters-degrees/unmanned-and-autonomous-systems-policy-and-risk-management-ms. [Accessed 5 March 2019].

[3] K. Kaur and G. Rampersad, "Trust in driverless cars: Investigating key factors influencing the adoption of driverless cars," *Journal of Engineering and Technology Management,* vol. 48, pp. 87-96, 2018.

[4] A. KIRILENKO, A. S. KYLE, M. SAMADI and T. TUZUN, "The Flash Crash: High-Frequency Trading," *THE JOURNAL OF FINANCE,* vol. LXXII, no. 3, pp. 967-998, 2017.

[5] A. Ollero, J. R. Martínez-de-Dios and L. Merino, "Unmanned Aerial Vehicles as tools for forest-fire fighting".

[6] V. Osadcuks and A. Galins, "SOFTWARE IN THE LOOP SIMULATION OF AUTONOMOUS HYBRID POWER SYSTEM OF AN AGRICULTURAL FACILITY," in *11th International Scientific Conference,* Jelgava, 2012.

[7] P. Koopman and M. Wagner, "Challenges in Autonomous Vehicle Testing and Validation," *SAE Journal on Transportation Safety,* vol. 4, no. 1, pp. 15-24, 2016.

[8] T. Menzies and C. Pecheur, "Verification and Validation and Artificial Intelligence," *Advances in Computers,* vol. 65, pp. 153-201, 2005.

[9] J. Schumann and W. Visser, "Autonmoy Software: V&V Challenges and Characteristics," 2006.

[10] P. Helle, W. Schamai and C. Strobel, "Testing of Autonomous Systems - Challenges and Current State-of-the-Art," in *26th Annual INCOSE International Symposium*, Edinburg, 2016.

[11] B. Davis and D. Lane, "Guided Construction of Testing Scenarios for Autonomous Underwater Vehicles Using the Augmented-Reality Framework and JavaBeans," *Journal of Engineering for the Maritime Environment,* vol. 224, no. M, pp. 173-191, 2010.

[12] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE JOURNAL OF ROBOTICS AND AUTOMATION,* Vols. RA-2, no. 1, pp. 14-23, 1985.

[13] G. Erann, "On Three-Layer Architectures," *Artificial Intelligence and Mobile Robots,* 1998.

[14] P. Milgram, H. Takemura, A. Utsumi and F. Kishino, "Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum," *SPIE Telemanipulator and Telepresence Technologies,* vol. 2351, pp. 282-292, 1994.

[15] J.-A. Fernandez and J. Gonzalez, "A FRAMEWORK FOR INTEGRATING THE SOFTWARE COMPONENTS OF A ROBOTIC VEHICLE," *IFAC Proceedings Volumes,* vol. 31, no. 2, pp. 419-423, 1998.

[16] S. Limsoonthrakul, M. N. Dailey, M. Srisupundit, S. Tongphu and a. M. Parnichkun, "A Modular System Architecture for Autonomous Robots Based on Blackboard and Publish-

Subscribe Mechanisms," in *ROBIO IEEE International Conference on Robotics and Biomimetics*, Bangkok, 2008.

[17] S. Tarkoma, PUBLISH/SUBSCRIBE SYSTEMS DESIGN AND PRINCIPLES, West Sussex: Jon Wiley & Sons Ltd, 2012.

[18] P. Eugster, P. Felber, R. Guerraoui and A.-M. Kermarrec., "The Many Faces of Publish/Subscribe," *ACM Computing Surveys,* vol. 35, no. 2, pp. 114-131, 2003.

[19] *Documentation - ROS Wiki,* http://wiki.ros.org/, 2019.

[20] *Master - ROS Wiki*, http://wiki.ros.org/Master, 2019.

[21] *What is Arduino?,* https://www.arduino.cc/en/Guide/Introduction, 2019.

[22] *Examples from Libraries,* https://www.arduino.cc/en/Tutorial/LibraryExamples, 2019.

[23] *roslaunch Package Summary - ROS Wiki,* http://wiki.ros.org/roslaunch, 2019.

[24] *rosserial Package Summary - ROS Wiki,* http://wiki.ros.org/rosserial, 2019.

[25] G. Cai, B. M. Chen and T. H. Lee, Unmanned Rotorcraft Systems, London: Springer-Verlag, 2011.

[26] B. Tribelhorn and Z. Dodds, "Evaluating the Roomba: A low-cost, ubiquitous platform for robotics research and education," in *Proceedings 2007 IEEE International Conference on Robotics and Automation* , Roma, 2007.

[27] J. Mcclean, C. Stullb, C. Farrarc and D. Mascareñas, "A Preliminary Cyber-Physical Security Assessment of the Robot Operating System (ROS)," in *Proceedings of SPIE - The International Society for Optical Engineering*, 2013.

VITA

Nathan Daniel Gonda **Graduate Researcher, Autonomous Systems**
Old Dominion University Dept. of Modeling, Simulation, and Visualization Engineering
5115 Hampton Blvd, Norfolk, VA 23529
ngond002@odu.edu

EDUCATION

Master of Science in Modeling and Simulation (GPA: 3.75) **Aug 2017 - Present**
Old Dominion University - Norfolk, Virginia
Relevant Courses: Machine Learning I and II, Intro to Combat Modeling & Simulation

Bachelor of Science in Modeling and Simulation (GPA: 3.94) **Aug 2014 - May 2017**
Old Dominion University - Norfolk, Virginia
Relevant Courses: Discrete Event Simulation, Continuous Simulation, Simulation Software Design, M&S Statistics & Analysis, Object Oriented Design, Introduction to Distributed Simulation, Linear Algebra

Associate of Arts and Sciences - Computer Science (GPA: 4.00) **Aug 2012 - May 2014**
Paul D. Camp Community College - Franklin, Virginia

PUBLICATIONS

- Gonda, N., Laverghetta, T.J., Leathrum, J.F., AN ARCHITECTURE FOR TEST & EVALUATION OF AUTONOMOUS SYSTEMS ALONG THE SPECTRUM OF MIXED REALITY. Paper submitted to the WinterSim Conference 2019, National Harbor, MD. (Submitted)
- Leathrum, J.F., Laverghetta, T.J., Gonda, N., Integrating Virtual and Augmented Reality Based Testing into the Development of Autonomous Vehicles. Paper in press at the MODSIM World Conference 2019, Norfolk, VA.
- Leathrum, J.F., Shen, Y., Mielke, R.R., Gonda, N., Integrating Virtual and Augmented Reality Based Testing into the Development of Autonomous Vehicles. Paper presented at the MODSIM World Conference 2018, Norfolk, VA.
- Collins, S.C., Gonda, N.D., Dumaliang, L.C., Leathrum, J.F., Mielke, R.R., (2017). VISUALIZATION OF EVENT EXECUTION IN A DISCRETE EVENT SYSTEM. Paper presented at the SpringSim-ANSS 2017 Conference, Virginia Beach, VA.
- Branch, B., Collins, S., Dumaliang, L., Gonda, N., Lane, T., Miles, K., Periman, M., Scerbo, D., Rapid USV Prototyping System (RUPS). Paper presented at the MODSIM World Conference 2017, Virginia Beach, VA.

AWARDS

VMASC Industry Association Undergraduate BS/MS Scholarship	March 2017
VMASC Industry Association VCCS Scholarship	July 2014
Gordon G. Barlow Jr. Memorial Scholarship	February 2013
Kiwanis Club of Smithfield Scholarship	February 2013
Paul D. Camp Classified Personnel Scholarship	August 2013
Louis Armstrong Jazz Award	June 2011