PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
GRADUATE PROGRAM IN COMPUTER SCIENCE

INTEGRATING ROBOT CONTROL INTO THE AGENTSPEAK(L) PROGRAMMING LANGUAGE

RODRIGO WESZ

Dissertation presented as partial requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Felipe Meneguzzi

Porto Alegre
2015

Dados Internacionais de Catalogação na Publicação (CIP)

W538i Wesz, Rodrigo

Integrating robot control into the Agentspeak (L) programming language / Rodrigo Wesz. – Porto Alegre, 2015.
132 p.

Dissertação (Mestrado) – Faculdade de Informática, PUCRS.
Orientador: Prof. Dr. Felipe Meneguzzi.

1. Informática. 2. Linguagens de Programação. 3. Robótica. I. Meneguzzi, Felipe. II. Título.

CDD 006.3

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "*Integrating Robot Control Into the Agentspeak(L) Programming Language*" apresentada por Rodrigo Buenavides Wesz como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 11/03/2015 pela Comissão Examinadora:

[illegible]

Prof. Dr. Felipe Rech Meneguzzi – Orientador — PPGCC/PUCRS

[illegible]

Prof. Dr. Alexandre de Morais Amoy – — PPGCC/PUCRS

[illegible]

Prof. Dr. Rafael Heitor Bordini – — PPGCC/PUCRS

[illegible]

Prof. Dr. Jomi Fred Hübner – — UFSC

Homologada em 28/08/2015, conforme Ata No. 015 pela Comissão Coordenadora.

[illegible]

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

INTEGRATING ROBOT CONTROL INTO THE AGENTSPEAK(L) PROGRAMMING LANGUAGE

ABSTRACT

Developing programs responsible for controlling mobile robots is not a trivial task. This led to the creation of several robot development frameworks to simplify this task. For each new rational behavior added to the robot, the number of events that the robot has to handle grows. Therefore, the development of the rational behaviors by using the frameworks may result in a source code which has more identifiers and large blocks of conditional statements, making difficult modularization and code reuse. This work presents a mechanism to program rational behaviors for mobile robots through the use of an agent programming language. This allows the robots programmer to develop rational behaviors using a higher level of abstraction in a modular fashion, resulting in simpler development and smaller, more readable and reusable code.

Keywords: robots, agents, rational behaviors, Jason, Cartago, ROS.

INTEGRAÇÃO DE CONTROLE DE ROBÔ NA LINGUAGEM DE PROGRAMAÇÃO AGENTSPEAK(L)

RESUMO

O desenvolvimento de programas para controle de robôs móveis não é uma tarefa trivial. Isso motivou a criação de vários frameworks para facilitar essa tarefa. Para cada novo comportamento racional adicionado ao robô, cresce o número de eventos que o robô tem de lidar, e desenvolver esses comportamentos racionais através do uso dos frameworks pode resultar em um código com mais identificadores e grandes blocos de condicionais, dificultando a modularização e reuso de código. Este trabalho apresenta uma forma de programar comportamentos racionais para robôs móveis através do uso de uma linguagem de programação de agentes. Isto permite ao programador de robôs o desenvolvimento de comportamentos racionais usando um nível de abstração mais alto e de forma modular, resultando em um desenvolvimento mais simples, e códigos mais legíveis, menores e reutilizáveis.

Palavras Chave: robôs , agentes, comportamentos racionais, Jason, Cartago, ROS.

LIST OF ACRONYMS

2APL	A Practical Agent Programming Language
3APL	Artificial Autonomous Agents Programming Language
AI	Artificial Intelligence
AMR	Autonomous Mobile Robots
AOP	Agent Oriented Programming
APL	Agent Programming Language
BDI	Beliefs Desires and Intentions
CARMEN	Carnegie Mellon Robot Navigation Toolkit
CARTAGO	Common Artifacts for Agents Open infrastructure
CCD	Charge-coupled device
CMOS	Complementary Metal-oxide Semiconductor
DMARS	Distributed Multi-Agent Reasoning System
FIPA	Foundation for Intelligent Physical Agents
GPS	Global Positioning System
ICE	Internet Communications Engine
IPC	Inter-Process Communication
JAL	Jack Agent Language
JADE	Java Agent DEvelopment Framework
JVM	Java Virtual Machines
LOC	Lines of code
MAS	Multi-Agent Systems
MOOS	Mission Orientated Operating Suite
MRPT	Mobile Robot Programming Toolkit
MVC	Model View Controller
OOP	Object Oriented Programming
ORCA	Organic Robot Control Architecture
PADI	Player Abstract Device Interface
PRS	Procedural Reasoning System
RADAR	Radio Detecting and Ranging
RMI	Remote Method Invocation
ROS	Robot Operating System
RDF	Robot Development Framework

SLAM – Simultaneous Localization and Mapping

SONAR – Sound Navigation and Ranging

UMVS – Marine Multivehicle Simulator

CONTENTS

1. INTRODUCTION

The development of technology allowed machines to replace humans in several tasks. From repetitive tasks at factory assembly lines, where pre-programmed robots operate in a simple, controlled environment; to hazardous tasks, such as work in mines, exploration in the deep sea where the pressure cannot sustain human life, and exploration in space, such as the Mars rover. Indeed, these machines became so popular nowadays that they are used for simple tasks, such as floor cleaning. These machines are called robots.

Various *robot development frameworks (RDF)* were created with the aim of facilitating the development of robotic behavior. These frameworks simplify the development of programs for robots kits, sensors and actuators. However, there are situations where to control the robot's hardware is not enough: in order to achieve a given goal, a robot may need to perform rational behaviors. For example: what should a housekeeper robot do first: answer the ringing phone or answer the knocking on the door? In order to allow the robot to have reasoning capabilities, such as choosing between the phone or the door, we need a computational analogy for the human reasoning and *rational agents* [37] have been successfully used in situations where modeling the human reasoning is necessary [10].

A rational agent is something that acts to achieve the best outcome or, at least, the best expected outcome, operating autonomously, perceiving its environment, persisting over a prolonged time period, adapting itself to changes, creating and pursuing goals [41]. To facilitate the development of agents, we have *agent programming languages (APL)* in which we are able to create agents that has some characteristics, such as *reactivity*: intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives; *pro-activeness*: intelligent agents are able to exhibit goal-directed behavior by taking the initiative in order to satisfy their design objectives; and *social ability*: intelligent agents are capable of interacting with other agents, or even humans, in order to satisfy their design objectives [55].

The development of rational behaviors using robot development frameworks may result in source code "polluted" by large blocks of if-then-else statements, jeopardizing modularization and code reuse. The motivation of this work is to simplify the modeling of complex behaviors in robots, allowing the development of mobile robots at a higher level of abstraction, through the use of agent programming languages. Thus, APL features can be explored to simplify the way robots are programmed, resulting in simpler development and smaller, more readable and reusable code.

There are two main advantages of choosing APLs to program mobile robots. First, the development process of any software using a higher abstraction level is simpler, faster, has better readability, better maintenance and more productivity comparing to the use of a low-level programming languages. Second, these languages follow the Beliefs-Desires-Intentions (BDI) architecture, based on the theory of human practical reasoning, developed by Michael Bratman [6]. Such model simplifies the development of rational behaviors, helping developers to program agents skilled to sense, reason and act in an autonomous way.

Our work integrates an agent programming language with a robot development framework through the use of an architecture responsible for connecting and communicating both technologies. The basic requirements of our architecture are:

- to program autonomous robots using a higher-level abstraction;
- to leverage agent behavior implementations into robots; and
- to simplify the development of complex behaviors on robots in a practical, extensible and scalable way;

1.1 Contributions

The main contributions of this work are first, we expanded Jason beyond the development of multi-agent systems, improving it to a high level mobile robot programming language; second, we simplify the development of rational behaviors by using a language that supports the BDI model, resulting in a simpler way of development programs for robots and improving code readability; and finally, a third contribution of our architecture is to improve code reuse and code modularization since our methods and classes are designed in a modular fashion way.

1.2 Publications

Part of this work resulted in a short paper and has been published as:
[51] Rodrigo Wesz and Felipe Meneguzzi. Integrating Robot Control into the AgentSpeak(L) Programming Language. Proceedings of the Eighth Workshop-School Agent Systems, their environments and applications, pages 197–203, 2014.

1.3 Structure

This work is organized as follows: Chapter 2 presents the basis of mobile robots, locomotion and perception; Chapter 3 introduces robot development frameworks commonly used by academic community to program mobile robots; Chapter 4 briefly introduces the concepts of rational agents; and finishing the theoretical background, the Chapter 5 shows the most common agent programming languages used to develop intelligent agents; Chapter 6 presents our integration architecture; Chapter 7 presents how the integration is implemented and how to develop our artifacts; Chapter 8 presents and discusses a set of experiments, comparing the use of our architecture with a non-agent programming language; Chapter 9 shows related work; and Finally, Chapter 10 presents the conclusions of our work and future work.

2. MOBILE ROBOTS

Autonomous Mobile Robots (AMR) are devices able to move autonomously through an environment, possibly performing previously programmed goal-oriented tasks. Autonomy means that the system can decide which action to take for itself independently [2]. In order to reach autonomy, the mobile robots need a set of characteristics [12] [2], and among these characteristics, those we consider most important are the following:

Mobility: the mobile robot must be able to choose and move to specific destinations in the environment without human direct external input [12] [2];

Adaptivity: AMRs will be confronted with situations which have not been specified before, and they need to adapt itself to these unknown situations [2];

Perception of the environment: the mobile robot must be able to retrieve information from the environment in order to infer the environment changes and take action based on these changes, such as navigate avoiding (or overcoming) obstacles [12] [2];

Knowledge acquisition: the mobile robot must have the ability to transform perception into knowledge while operating [2];

Real-time processing: all the previous features require an architecture able to handle the continual input, process and output of data. The data requests should be dealt quickly enough to enable the robot to take immediate action when needed [2];

To be able to move through the environment and to perceive the environment, basic requirements are needed at the physical hardware perspective. The architecture should include at least: a power source to be carried along with the mobile robot; actuators to make the mobile robot able to move through its environment; and a collection of sensors which the robot collects information from the environment [12]. In the following sections, we briefly review key concepts regarding power supply, locomotion and perception.

2.1 Power Supply

Without a power supply, a robot is nothing more than a piece of static hardware. Mobile robots commonly use electric batteries as power source, in which chemical reactions generate electrical energy for a certain amount of time. Mobile robots use motors to convert the electrical energy obtained from the batteries to mechanical energy in order to move themselves. Batteries get discharged gradually while in use, but they can be recharged by providing an electrical current to it. Eventually, batteries achieve a point at which they cannot be recharged, and must be replaced [12]. Alternatively, the power may be supplied by fuel cells (chemical reaction by the combination of hydrogen and oxygen to produce electricity), compressed gases, gasoline or other fuels. [44]

The power supply must provide the amount of required power for all the robot's components at all times, but different components require different voltages and most batteries are never at a constant voltage [12]. A battery may be over its nominal voltage when fully charged and may drop to 50% of its nominal voltage when drained, depending on its chemistry [26], this variability may generate problems in components that are sensitive to the input voltage, such as microcontrollers restart due to lack of power or erroneous sensors readings. To avoid such kind of problems, fuses, capacitors, voltage regulators and switching regulators are used in the power supply regulation [26].

2.2 Locomotion

Locomotion is the process by which an autonomous robot moves. In order to produce motion, forces must be applied by the mobile robot [12]. To model the motion solution, we must consider two aspects: *kinematics* and *dynamics*.

Kinematics is the study of motion without considering the cause of the motion. It does not regard the forces that affect the motion, but just the motion mathematics[1] [12]. Also known as geometry of motion, kinematics studies the motion of points, lines, objects, groups of objects and properties such as velocity and acceleration [20]. For wheeled robots, kinematics is enough to cover locomotion tactics. This kind of robot takes advantage of the ground contact and friction to move from one place to another. They can use wheels, tracks or limbs. The wheel is the oldest and most popular locomotion mechanism in general man-made vehicles, as well as in mobile robots [45].

Dynamics is the study of forces[2] and torques, their effect on the motion, the energy and the speed required by the motion and the causes of motion. For legged, aquatic and aerial robots, consider the dynamics is usually necessary to cover locomotion tactics [12]. Aquatic robots take advantage of the surrounding water to make the propulsion needed to move from one place to another while aerial robots use the same aerodynamics used by planes and helicopters to make movement [45].

2.3 Perception

Perception is the process by which an autonomous robot acquires knowledge about itself and its environment. In order to perceive, the robot must extract meaningful information from sensor measurements [12]. A sensor is a device that detects or measures a physical property and converts it into human-readable display or into a signal which can be read and further processed [20]. Sensor attributes must be considered when modeling the architecture of a given mobile robot project: sensors are noisy, return an incomplete description of the environment and cannot reconstruct the real world in all its fidelity [12]. According to Siegwart et al. [45] there are characteristics which

[1]Kinematics can be abstracted into mathematical functions such as planar algebra and unit circle [12].
[2]Isaac Newton [31] defines force as an exertion or pressure which can cause an object to move [20].

determine how close to reality the information coming from the sensors are and these characteristics are the following:

- *Dynamic range*: are used to measure the difference between the upper and lower limits of the sensor input values [22]. This helps to define how the sensor reacts when exposed to input values above or below its working range [45].
- *Linearity*: is related to the behavior of the sensor's output signal as the input signal varies. There is a function that can be used to predicts the expected output from the sensor, given the input from the environment. The measurement ratio must be constant with the function results [45].
- *Frequency*: is the speed at which a given sensor can perform a stream of readings [22]. Commonly, the number of measurements per second is defined in hertz [45].
- *Sensitivity*: is the measure of the degree to which a change in the input signal outcomes in a change in the output signal [45]. In other words, it is the minimum input signal magnitude needed to generate a distinct output signal [22].
- *Cross-sensitivity*: is an undesirable characteristic on which a sensor detects a physical property that it is not interested in. It happens when the sensor gather a property from the environment that are orthogonal to its target property [45].
- *Resolution*: is the smallest change detectable by a sensor in a given measuring unit [22]. In other words, it is the minimum difference between two values that a sensor can gather from the environment [45]. It is not the same as sensitivity, since resolution is related to the ability of distinguish between closely adjacent values of a certain measuring unit.
- *Accuracy*: is the degree of conformity between what is gather from the environment by the sensor and the true value from the environment [22]. In other words, it is the degree of closeness between the measured value and the true value [45].
- *Precision*: is not the same as accuracy. It is related to the reproducibility of the sensor results when the measure is made on the same input [45]. Thus, when repeated measurements under the same conditions generate the same results, we have a precise sensor [22].

Depending on the environment, distinct problems need to be handled. For example, after the operational environment is chosen, robot's kinematics and dynamics need to be modeled. Problems such as the relation between the velocity of the wheels, the robot movement and wheels slippage needs to be addressed when the environment is overland. On any kind of environment, problems such noise and incomplete information from the sensors need to be considered in order to avoid obstacles.

3. ROBOT DEVELOPMENT FRAMEWORKS

The development of algorithms able to control mobile robots is not a trivial task and, as presented in Section 2, a set of ordinary requirements are needed and distinct problems must to be handled: for each requirement and problem, we need to develop a algorithm (or a set of algorithms) responsible for handling with it. Such characteristics motivated the development of many software systems and frameworks which make robot programming easier. This chapter presents a brief introduction about the development frameworks commonly used by the academic community to program mobile robots.

3.1 Robot Operating System

The Robot Operating System (ROS) [34] is not an operational system in the traditional sense - despite what the name suggests - but a framework created with the objective of helping software developers to create applications for robots. ROS provides hardware drivers, simulators, inter-process communication and allows code reuse. The fundamental concepts of ROS implementation are nodes, messages, topics, and services:

Nodes are processes that perform some kind of computation. Namely, a node is an instance of an executable which may communicate with each other by passing *messages* or through the use of a *service*. ROS is designed to be modular at a fine-grained scale and in this context, the term node is interchangeable with software module [34]. Nodes can connect to each other directly or find other nodes to communicate using a lookup service provided by a special ROS node called *Master*. Thus, nodes can declare themselves to the Master node, which acts as a name service and makes the nodes able to find each other, exchange messages or invoke services.

A *Topic* is a data transport system, based on the publisher-subscriber pattern [13]. Nodes send messages by publishing on a topic and nodes receive messages by subscribing to a topic. There may be multiple concurrent publishers and subscribers for a single topic as shown in Figure 3.1. Although the topic-based publish-subscribe model is a flexible communication paradigm, its broadcast routing scheme is not viable for synchronous transactions, which can simplify the design of a given node [34]. Synchronous transactions can be obtained through the use of *services*.

A *Message* is simply a data structure. It can be composed of other messages, and arrays of other messages, nested arbitrarily deep [34]. A node sends a message by publishing it on a given topic and a node that is interested in a certain kind of data subscribes to the appropriate topic. For example, a node representing a robot servo motor may publish its status on a topic with a message containing an integer field representing the position of the motor, a floating point field representing its speed and any other additional information relevant for its functionality.

A *Service* is the synchronous communication method used in ROS. It is analogous to web services, which are defined by URLs and have request and response messages of well-defined types

[34]. Note that, unlike topics, only one node can advertise a service of any particular name: there can only be one service called `get_image`, for example, just as there can only be one web service at any given URI.

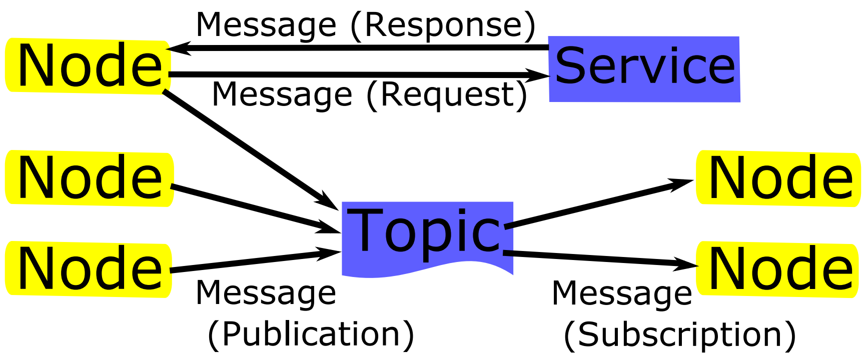

Figure 3.1 – Base concepts of ROS [14].

Figure 3.2 illustrates an example of the ROS communication mechanism: in this example, there are three nodes: the ROS `Master` node, a `Camera` node and an `Image viewer` node. The ROS `Master` node provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of ROS `Master` node is to enable individual ROS nodes to locate one another [47]; the `Camera` node is responsible for publishing images on a topic; and the `Image viewer` node needs to receive the information from the camera. On this example, these nodes are using a topic called `images`.

On step 1, the `Camera` node notifies ROS `Master` node that it wants to publish images on the `images` topic. After ROS `Master` node recognizes the `Camera` node, it may publish images to the `images` topic, but there are no clients subscribing to that topic yet, thus no data is actually sent. On step 2, `Image viewer` node wants to subscribe to the `images` topic in order to gather information from the camera, thus it notifies ROS `Master` node that it wants to receive images from `images` topic. On step 3, the `images` topic has both a publisher and a subscriber, thus ROS `Master` node notifies `Camera` node and `Image viewer` node about each other's existence, then `Camera` node can start transferring images to the `images` topic (red bubble 1) and `Image viewer` node starts receiving images from the topic (red bubble 2).

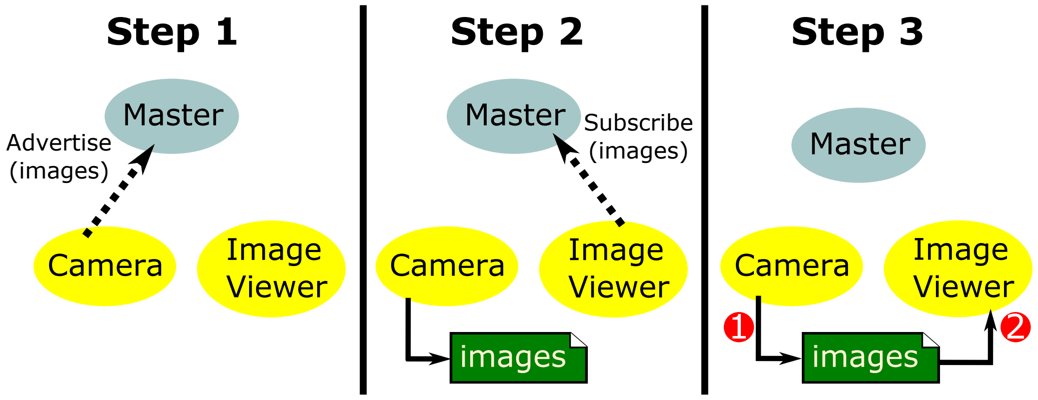

Figure 3.2 – ROS communication mechanism example [47].

3.2 Carnegie Mellon Robot Navigation Toolkit

The Carnegie Mellon Robot Navigation Toolkit (Carmen) is a repository of algorithms, libraries and applications for mobile robot control and a set of simulators for mobile robots platforms [29] maintained by Carnegie Mellon University. The project's objective is to facilitate the development and sharing of algorithms between different research institutions, in addition to establishing good programming practices. To reach these objectives, Carmen was designed with each component being implemented as a separate module. Each module is responsible for one task, avoiding multiple features into a single module, making the code maintenance easier and taking advance of flexibility, extensibility and reliability performed by the use of modularization. Native modules can be replaced by third-party modules and the modules can be executed remotely instead of on-board the robot. Carmen is written in C and has a three-tiered architecture: The lowest layer controls hardware interaction (sensors and actuators); the second layer handles the navigation in the environment: including motion planning and barrier search, detection and avoidance; and the third layer is the interface with the user.

Carmen uses the model-view-controller (MVC) design pattern [13], which separates the data from the user interface, in this way, all the process communication is done by interfaces. If a maintenance or upgrade is needed on the communication protocol, only the code in the outbound and inbound interface libraries must be changed, preserving the core code unchanged. Carmen stores the parameters used in the modules in a single place: the Centralized Repository, thereby ensuring that all modules have the same information avoiding conflicts that may occur when parameters are loaded from different locations.

The Carmen's module communication is performed by a version of Inter-Process Communication (IPC) [46] developed at Carnegie Mellon University. This IPC version is a publish/subscribe model: a process needs to subscribe to receive messages from an event or from another process, or to receive messages of a certain type. All subscribers asynchronously receive a copy of a certain message. IPC uses a central server to route and log the messages. This server keeps information about all the system. The processes must connect to the server in order to send or receive messages.

In order to navigate, Carmen uses a navigation function to generate a gradient field that represents the optimal (lowest-cost) path to the goal at every point in the workspace by continuously calculating an optimal path to a goal. The Carmen navigation is accomplished by tracking differences between actual scans (performed by the planner) and expected laser scans (from the maps repository) given the most likely position of the robot at every time-step in a non-static or incompletely known environment. This function is called gradient descent planner [24].

3.3 Mission Orientated Operating Suite

Mission Orientated Operating Suite (MOOS) [30] is an extensible C++ middleware for mobile robotics research designed by the department of Ocean Engineering at Massachusetts Institute of Technology. It has been designed in a modular way, allowing developers to keep coding their own application independently of all other source code, thus code change in one domain cannot cause compilation failures in another domain [30]. It is composed of two parts: a set of libraries and applications to control a field robot in sub-sea and land domains and a simulator called Marine Multivehicle Simulator (uMVS) which allows the simulation of an environment with any number of vehicles. The features of this simulator are vehicle dynamics simulation, center of gravity simulation, acoustic simulation and velocity dependent drag [30].

MOOS has a star topology: each application (called MOOSApp) has a connection to a single MOOS Database (called MOOSDB) thus, all communication happens via this central server application; there is no app to app communication. The database never makes an unsolicited attempt to contact an app, therefore all communication between the client and server is tempted by the client and a given client does not need to know that another client exists [30].

There are three kinds of communication mechanisms on MOOS: *data publishing* from an app, *registering* for notifications and *notifications collecting*. *Publish data* is the simplest way a MOOSApp has to transmit its data. I works as the following: MOOSApp data is sent to the MOOSDB and MOOSDB is responsible for delivering the message (if needed); The *registration mechanism* is used by a MOOSApp to register its interest in a certain data. It works like a subscription, in which the client receives a collection of messages describing each and every change notification issued on a given data; And the mechanism of *collect notifications* is used by the MOOSApp to inquire whether it has received any new notifications from the MOOSDB.

MOOS has two types of navigation: manual, controlled by a module called `iRemote` and automatic, controlled by `pHelm` and `pNav` modules. In the manual navigation, the `iRemote` method allows the remote control of the vehicle's actuators. Each actuator has a specific keyboard's key responsible for activating it [30]. For example, to move the vehicle left or right, the pilot must press N or M keys respectively; and in the auto navigation, the `pHelm` method is responsible for deciding the most suitable actuation command to be executed and the `pNav` method is responsible for moving the vehicle, controlling its speed and direction [30].

3.4 Organic Robot Control Architecture

The Organic Robot Control Architecture (Orca) is an open-source framework for robotic systems development. This project aims to maintain a community in which code is shared between different research groups in order to improve the progress of robotic research and the robotic industry [28]. It was created to achieve requirements such as [28]: enable software reuse by defining a set of commonly-used interfaces; simplify software reuse by providing libraries with a high-level convenient API; encourage code reuse by maintaining a repository of components; be general, flexible and extensible; be sufficiently robust, high-performance and full-featured in order to be used in commercial applications; and be sufficiently simple for experimentation in university research environments, in which the last two items are somewhat contradictory and not easily combined [28].

Orca provides the means for developing independent code blocks to be used together in order to achieve a goal. The goal may range from complex robotics systems to decentralized sensors networks [28]. Orca's code reuse is based on supplying developers with a high-level API, setting a group of commonly-used interfaces and sustaining of a repository of components. Orca aims to be as broadly applicable as possible, therefore no assumptions about component granularity, system architecture, required interfaces and internal architecture of components was made [28].

The communication mechanism of Orca is provided by the Internet Communications Engine (Ice) [19], using a protocol that supports both TCP and UDP. Ice optionally provides a flexible thread pool for multi-threaded servers, has libraries to several languages (such as C++, Java, Python, PHP, C#, and Visual Basic) and builds natively under several operating systems (such as Linux, Windows, and MacOS X) [28].

3.5 Player

The Player project is composed of tools in whose objective is to simplify and speed up the development of code to control robots by creating an abstract interface for robot hardware, based on network engineering and operating systems techniques [48]. It consists of two main products: Player, an interface to a collection of hardware devices and Stage[1], a virtual robot device simulator.

The Player core - responsible for enabling portability and code reuse - is called Player Abstract Device Interface (PADI). PADI defines a set of interfaces that capture the functionality of logically similar sensors and actuators. The code reuse and portability is achieved because PADI defines the data semantics and data syntax that is exchanged between the robot hardware and the code that controls the robot, allowing Player-based controllers to execute unchanged on a variety of real and simulated devices [48]. There are three abstractions that underlie Player project: the *character device* model, the *interface/driver* model, and the *client/server* model.

The character device model is part of the device-as-file model used on UNIX operating system. It establishes that all I/O devices can be thought of as data files. There are two types of devices: *random access devices*, also known as block devices, which are able to manipulate chunks of data in arbitrary orders, such as disk drives and *sequential access devices*, also known as character devices, able to manipulate streams of data one byte after another, such as tapes and terminals. Sensors and actuators are character devices and Player uses the character device model operations to access its hardware devices [48]. In order to gather information from a given sensor, Player uses

[1]The Stage simulator is presented in Section APPENDIX D

an *open* operation followed by a *read* operation. To control a given actuator, Player uses an *open* operation followed by a *write* operation.

The interface/driver model is responsible for determining the content of the data streams used by the character device model, providing device independence and it is responsible for grouping the devices by logical functionality, thus devices that do almost the same job will appear identical to the developer [48]. One limitation of having a generic interface for entire classes of devices is that special features or functionalities of a given device are ignored. The developer has two ways for addressing this limitation: he may implements a new interface for these kind of devices including the support of the special feature, or the developer may adds the special feature to the existing class.

The client/server model is responsible for providing a practical architecture for implementing a robot interface [48]. Player is based on a client/server model, in which the server is responsible for executing low-level device control and the client is the program controlled by the developers. One advantage of this model is that the client can be written in any programming language compatible with a TCP socket [48]. This makes Player almost language-neutral because TCP is supported by almost every modern programming language, and it enables clients to be executed on any machine with network connectivity to the server machine. Two disadvantages of this model are that it introduces additional latency to device interactions since all data and commands need to pass through the server and it requires network hardware and software.

3.6 Considerations

A number of elements contribute to an increase of interest in robot programming, such as advances in technology, low-cost of basic robotic hardware and software/firmware availability. These elements motivated the creation of various robot development frameworks, such as the RDFs presented in this Chapter.

From robot arms at factory assembly lines to homemade robots created using technologies such as Arduino[2], we need a control layer responsible to interact with hardware. This control may be provided by the RDFs, which are able to handle robot kits, sensors and actuators.

This Chapter presents RDFs commonly used to control autonomous mobile robots, i.e, robots able to move autonomously through an environment, possibly performing previously programmed goal-oriented tasks. AMRs are being increasingly employed in real-world applications and the development of algorithms able to control mobile robots is not a trivial task, making this area of high interest to the academic community.

[2]More information about Arduino is available at http://www.arduino.cc/

4. RATIONAL AGENTS

The study of artificial intelligence started after the World War II, and the term was coined in 1956. Russell and Norvig [41] define AI as *"the study of agents that receive percepts from the environment and perform actions"*, but a huge number of different definitions was created since the fifties: *"AI . . . is concerned with intelligent behavior in artifacts."* Nilsson [32]; *"The study of the computations that make it possible to perceive, reason, and act."* Winston, 1992 [53]; *"The exciting new effort to make computers think . . . machines with minds, in the full and literal sense."* Haugeland [18]; on all these definitions, the main unifying theme is the idea of a rational agent [41]. According to Haugeland [18] and Norvig et al. [41], an agent is just something that perceives and acts. On the other hand, a rational agent acts to achieve the best outcome or, at least, the best expected outcome, operating autonomously, perceiving its environment, persisting over a prolonged time period, adapting itself to changes, creating and pursuing goals [41]. According to Wooldridge [55], a rational agent should have three properties: *reactivity*: intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives; *pro-activeness*: intelligent agents are able to exhibit goal-directed behavior by taking the initiative in order to satisfy their design objectives; and *social ability*: intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.

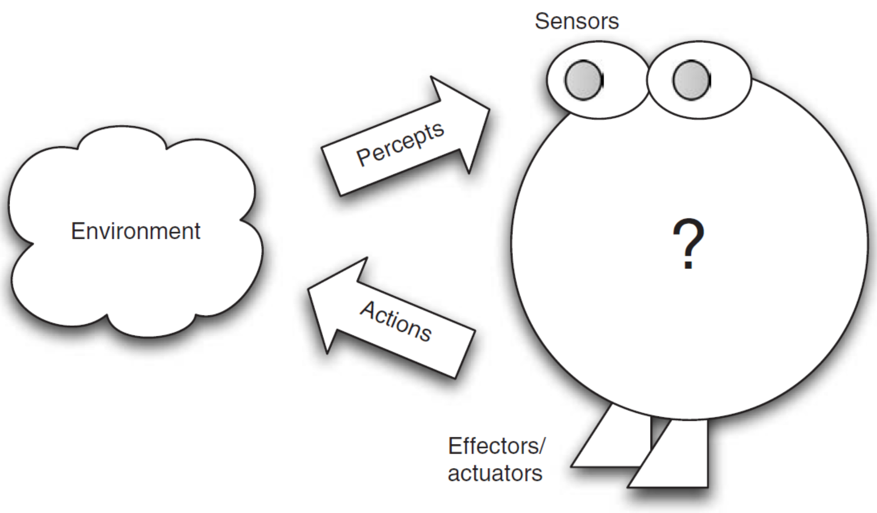

Figure 4.1 – An agent and its environment [41].

As Figure 4.1 illustrates, an agent is able to interact with an environment (which may be physical, in case of robots on the physical world or may be a software, in case of a simulator), and an agent is capable of sensing the environment via sensors. As the agent decides what to do based on the information obtained via these sensors, the set of possible actions to be performed by the agent - in order to modify the environment - depends on the information perceived from these sensors. When a given action is chosen, the agent modifies the environment through the use of the actuators and effectors. In an multi-agent system, the same environment is shared by multiple agents, each of them acting on the environment, interfering and communicating with the other agents.

4.1 Agent Architectures

The essential operation of gathering information from the environment (through sensors) and selecting and executing the best action to apply (through actuators) depends on the agent's *architecture*, and this architecture is composed by a collection of software (or hardware) modules which define how the sensor data and the current internal state of the agent determine the agent's actions [56]. According to Wooldridge and Jennings [56], there are three different approaches to building agents: the classical approach (deliberative architecture); the alternative approach (reactive architectures); and the hybrid approach (hybrid architecture).

Deliberative Architectures: An agent built using this architecture realizes deliberation about alternative courses of action before any action is taken, thus, it is focused on long term planning of actions. Wooldridge and Jennings [56] define a deliberative architecture as one that contains an explicitly represented, symbolic model of the world, and in which decisions (for example about what actions to perform) are made via logical (or at least pseudo-logical) reasoning, based on pattern matching and symbolic manipulation. As Figure 4.2 illustrates, based on its word model, the agent applies to an action through the use of planning operations.

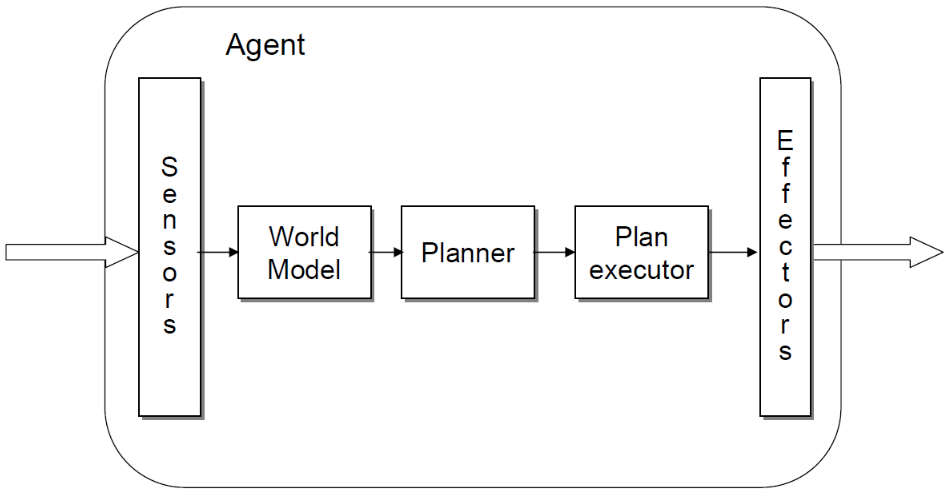

Figure 4.2 – Basic Schema of a Deliberative Agent [54].

Reactive architectures: An agent built using this architecture is based on fast reactions to the changes detected in the environment. Wooldridge and Jennings [56] define a reactive architecture as one that does not include any kind of central symbolic world model, and does not use complex symbolic reasoning, instead, it connects sensory inputs to specific actions. It should be stressed that the resulting systems are, in terms of the amount of computation they need to do, extremely simple, with no explicit reasoning of the kind found in symbolic AI systems [56]. As Figure 4.3 illustrates, there is no world model, no planning and for each state, there is an action to be performed.

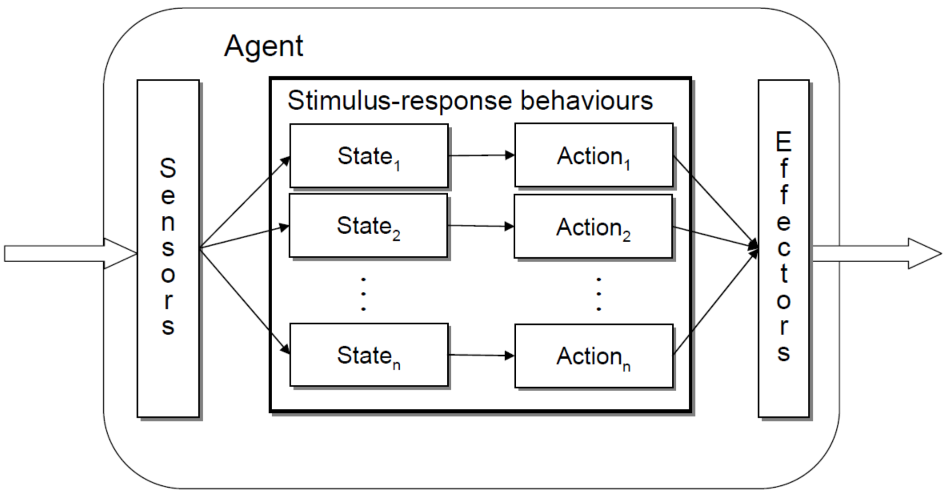

Figure 4.3 – Basic Schema of a Reactive Agent [54].

Hybrid Architectures: An agent built using this architecture is based on a combination of deliberative and reactive architectures. According to Wooldridge and Jennings [56], the deliberative subsystem contains a symbolic world model, develops plans and makes decisions in the way proposed by mainstream symbolic AI; and the reactive subsystem is capable of reacting to events that occur in the environment without engaging in complex reasoning. Often, the reactive component has some kind of precedence over the deliberative one, so that it can provide a rapid response to important environmental events [56]. Figure 4.4 illustrates the basic schema of a hybrid agent.

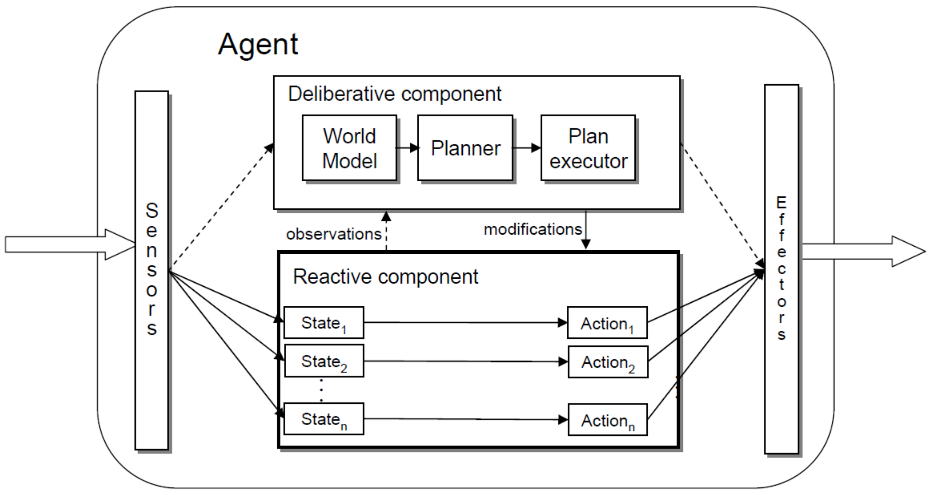

Figure 4.4 – Basic Schema of a Hybrid Agent [54].

4.2 Belief-Desire-Intention Model

The Belief-Desire-Intention (BDI) model is a theory of human practical reasoning, developed by Michael Bratman [6]. This philosophical model, inspired by human behavior, is the base of the BDI software model developed for programming intelligent agents and it provides a computational analogy for human practical reasoning. *Beliefs* of an agent are what the agent considers to be true regarding its current environment and a belief is an expectancy, it is not necessarily accurate; *Desires* are a set of states that the agent would like to achieve and these states may be mutually inconsistent; *Intentions* are the states that the agent chose to achieve: The agent selects a course of action and become committed to this course in order to reach that states [6].

The process of practical reasoning in a BDI agent are composed by beliefs, desires and intentions, in addition to a *belief revision function*, an *option generation function*, a *filter function* and an *action selection function* [50], as shown in Figure 4.5. *The belief revision function* first gather information from the environment; after that checks the current agent's beliefs set; then, on the basis of these, determines a new set of beliefs; The *option generation function* is responsible for determining the options (namely, the viable desires) available to the agent, and it is based on its current beliefs, its environment and its current intentions; The *filter function* chooses the agent's intentions based on its current beliefs, desires, and intentions; and the *action selection function* determines an action to perform based on its current intentions [50].

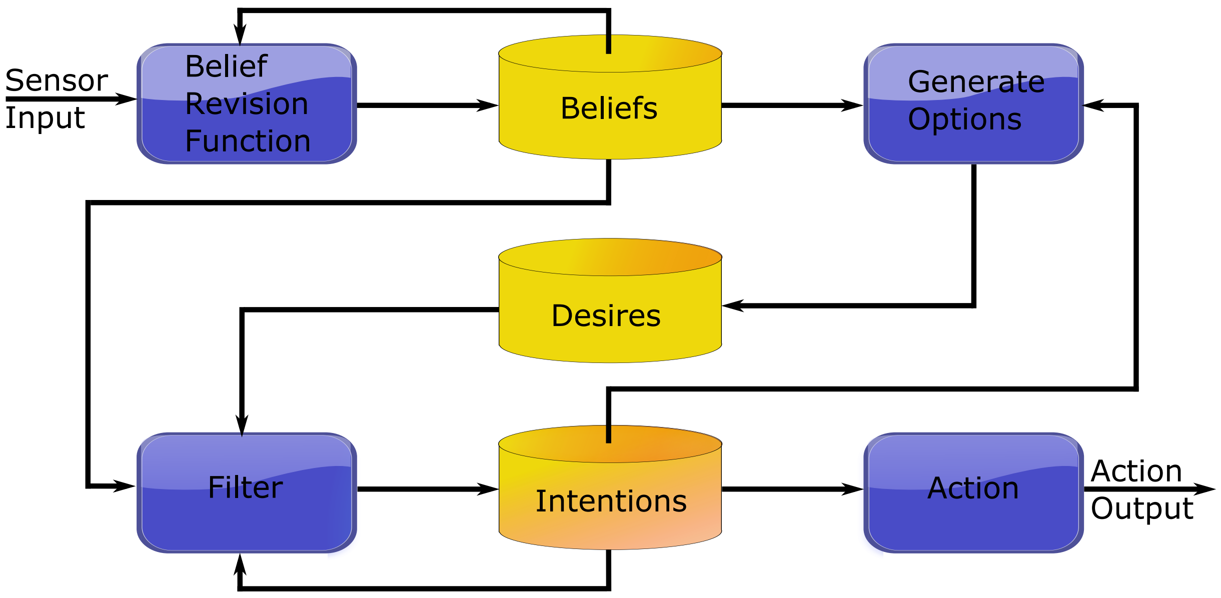

Figure 4.5 – Belief-desire-intention architecture [50]

To support the BDI model, some programming languages were created, such as AgentSpeak(L) [35], Jason [5] and 2APL [9]. Through the use of agent programming languages that support the BDI model, developers are able to program agents skilled to sense, reason and act in an autonomous way. Section 5 presents a set of agent programming languages commonly used by the academic community to program BDI agents.

5. AGENT PROGRAMMING LANGUAGES

The growing research in Multi-Agent Systems (MAS) has led the development of programming languages and tools which help the implementation of these systems [3]. Surveying the MAS literature will reveal a large number of different proposals for Agent Programming Languages (APL), such as APLs which support the BDI model. The use of APLs, instead of a more conventional programing language, proves useful when the problem is modeled as a multi-agent system, and understood in terms of cognitive and social concepts such as beliefs, goals, plans, roles, and norms [3]. In this Chapter, we first briefly introduce the Agent-Oriented Programming paradigm; second, we introduce AgentSpeak(L) and its extended language Jason; then, we present a brief introduction about other APLs that support the BDI model commonly used by the academic community; and finally, we present a brief introduction about multi-agent environments.

5.1 Agent-Oriented Programming

The term agent-oriented programming (AOP) was coined by Yoav Shoham [43] to describe a programming paradigm based a societal view of computation, in which multiple agents interact with one another, inspired by previous research in distributed and parallel programming and AI. According to Shoham [43], from the engineering point of view, AOP can be viewed as a specialization of the object-oriented programming (OOP) paradigm: whereas OOP proposes viewing a computational system as made up of modules that are able to communicate with one another and that have individual ways of handling incoming messages, AOP specializes the framework by fixing the state (now called mental state) of the modules (now called agents) to consist of components such as beliefs, capabilities, and decisions, each of which enjoys a precisely defined syntax. Table 5.1 summarizes the relation between AOP and OOP. A complete AOP system includes three primary components [43]:

- a *restricted formal language* with clear syntax and semantics for describing mental state. The mental state will be defined uniquely by several modalities, such as belief and commitment;
- an *interpreted programming language* in which to define and program agents, with primitive commands such as REQUEST and INFORM. the semantics of the programming language will be required to be faithful to the semantics of mental state; and
- an *agentifier*, converting neutral devices into programmable agents.

Since imperative languages are not the best choice for expressing the high-level abstractions associated with agent systems design, the research community has created several agent programming languages that help the development of agents and multi-agent systems. The next Sections present agent programming languages commonly used by the academic community.

	OOP	AOP
Basic Unit	object	agent
Parameters Defining State of Basic Unit	unconstrained	beliefs, commitments, capabilities, choices...
Process of Comunication	message passing and response methods	message passing and response methods
Types of Message	unconstrained	inform, request, offer, promise, decline...
Constraints on Methods	none	honesty, consistency...

Table 5.1 – OOP versus AOP [43].

5.2 AgentSpeak(L)

AgentSpeak(L) is a logical programming language for the implementation of BDI agents. It is based on a restricted first-order language with events and actions. AgentSpeak(L) can be viewed as a simplified textual language of Procedural Reasoning System (PRS) [36] or Distributed Multi-Agent Reasoning System (dMARS) [11] [35].

PRS [36] was one of the first implemented systems to be based on the BDI architecture [50]. It was implemented in LISP and provides goal-oriented behavior and reactive behavior. PRS was created to support real-time malfunction-handling and diagnostic systems. It has been used for a wide range of applications in problem diagnosis such as the Space Shuttle, air-traffic management, and network management [50]. dMARS [11] is a faster, more robust reimplementation of PRS in C++. It has been used in a variety of operational environments, such as air combat simulation, resource exploration and malfunction handling on NASA's space shuttle [50].

5.2.1 Introducing the AgentSpeak(L) Programming Language

This section introduces the AgentSpeak(L) language syntax. The alphabet of the formal language consists of variables, constants, function symbols, predicate symbols, action symbols, connectives, quantifiers, and punctuation symbols.

In addition to first-order connectives, the AgentSpeak(L) language use ! (for achievement), $?$ (for test), ; (for sequencing), and $\leftarrow$ (for implication). In the agent programs, AgentSpeak(L) use $\&$ for $\wedge$, not for $\neg$, <- for $\leftarrow$ and, such as Prolog[1] - AgentSpeak(L) requires that all negations be grounded when evaluated. It uses the convention that variables are written in upper-case and constants in lower-case, an underscore sign for an anonymous variable, standard first-order definitions of terms, first-order formulas, closed formulas, and free and bound occurrences of variables.

[1]Prolog is a general purpose logic programming language that is used for solving problems that involve objects and the relationships between objects [8].

Beliefs represent the agent's view of the current state of the environment (including other agents in the environment). They are defined as follows [35]: if p is a predicate symbol, and $t_1, ..., t_n$ are terms, then $p(\vec{t})$ is a belief atom, in which $\vec{t}$ is a (possible empty) list of terms. If $p(\vec{t})$ and $q(\vec{u})$ are belief atoms, $p(\vec{t}) \wedge q(\vec{u})$, $\neg p(\vec{t})$, and $\neg q(\vec{u})$ are beliefs. Terms in AgentSpeak(L) can be either constants or variables, and follow the Prolog convention [8].

Desires, in the BDI literature, are states that the agent wants to reach. Although the term desire is not used in its original definition [35], AgentSpeak(L) implicitly considers goals as adopted desires. There are two types of goals: achievement goals and test goals. Achievement goals express which state the agent wants to achieve in the environment and test goals verify if the predicate associated with the goal is true. Test goals either succeed if they unify with a belief in the belief base, or fail if it is not possible to unify with any agent belief. In the syntax, goals are defined as follows. If p is a predicate symbol, and $t_1, ..., t_n$ are terms, then $!p(\vec{t})$ is an achievement goal, and $?p(\vec{t})$ is a test goal, in which $\vec{t}$ is a (possible empty) list of terms.

Intentions in AgentSpeak(L) are plans adopted to satisfy the agent's goals. In order to satisfy these goals, an AgentSpeak(L) agent uses *plans* from a *plan library*. A plan library contains a set of conditional plans that can be used to respond to *events*, which results in the agent executing *actions* in the environment.

Events in AgentSpeak(L) can be either externally perceived, when a change in the environment is perceived, or internally generated when a new goal is adopted, or explicit additions or deletions of beliefs are generated by the agent. These events are called *triggering events*, and are represented as follows: to remove a belief or a goal, AgentSpeak(L) language uses the "-" operator; to add a belief or goal, the "$+$" operator is used; if $b(a)$ is a belief and $!g(a)$ and $?g(a)$ are goals, then $+b(a)$, $-b(a)$, $+!g(a)$, $-!g(a)$, $+?g(a)$, $-?g(a)$ are triggering events. [35].

Based on the environment observation and its own goals, an AgentSpeak(L) agent may desire to change the state of the environment and it is done by executing actions. If a is an action symbol and $t_1, ..., t_n$ are terms, then $a(\vec{t})$ is an action, in which $\vec{t}$ is a (possible empty) list of terms [35].

Now that we know the concept of a triggering event and an action, we can talk about *plans*. A plan consists of a head and a body. The head is composed of a triggering event - which specifies what causes the plan to be triggered - and a context condition, which specifies the minimum requirements for the plan to execute successfully. The body consists of a sequence of steps that an agent must accomplish to achieve the goal associated with the triggering condition. These steps can be actions that the agent executes in the environment or goals that the agent tests or achieves. If e is a triggering event, $b_1, ..., b_m$ are belief literals, and $h_1, ..., h_n$ are plan steps then $e : b_1 \wedge ... \wedge b_m \leftarrow h_1; ...; h_n$ is a plan [35]. Thus, a plan is adopted only when an event occurs that matches its triggering condition and the context condition holds in the agent's belief base. More specifically, a plan in the plan library is *relevant* to respond to an event if its triggering event matches the event, and *applicable* if the context condition holds. In summary, an agent is composed of a belief base, a set of events, a set of actions, a plan library and a set of intentions.

5.2.2 Interpreter Cycle

A program developed to create an agent or a multi-agent system has a different behavior from a conventional programming language such as Java or C. AgentSpeak(L) executes inside a loop: the language interpreter has a set of steps and when it reaches the last step, it starts from the beginning again. Figure 5.1 shows the interpretation cycle of an AgentSpeak(L) program: beliefs, events, plans and intentions are represented by rectangles, the selection of an element from a set is represented by rhombuses and processes are represented by circles.

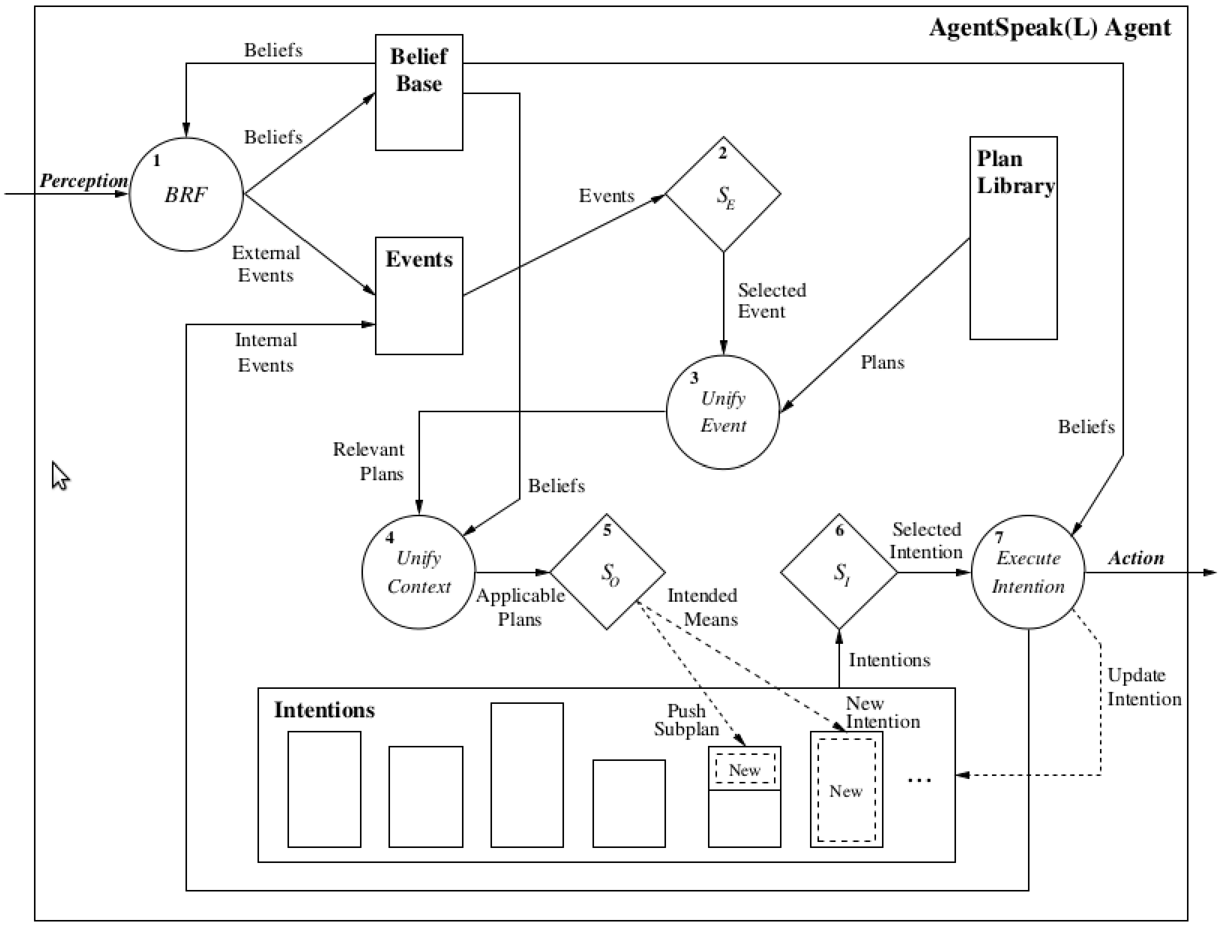

Figure 5.1 – Interpretation Cycle of an AgentSpeak(L) Program [27].

When an internal or an external event happens, an appropriate triggering event is generated and the event (internal or external) is asynchronously added to the set of events. After that, the *function SE* (Figure 5.1, rhombuses 2) selects an event from the set of events, remove it from the list, and use it to unify with the triggering events of the plans on the plan library. There may be many applicable plans for each event and the *function S0* (rhombuses 5) is responsible for choosing a single applicable plan from the plan library. If the event is an external one, S0 function creates a new intention stack in the set of intentions. If the event is internal, then the function S0 pushes that plan on the top of an existing intention stack, as shown in figure 5.1.

The next step is performed by the *function S1*: it selects a single intention to be executed, as shown in figure 5.1, rhombuses 6. The intention is a stack of partially instantiated plans, and when an intention is executed, the first goal or action on the body of the top of the intention stack is executed. When an action is executed the action is added to the set of actions and is removed from the body of the top of the intention. Finally, the agent goes to the set of events and the cycle repeats until there are no events in the events list or there is no runnable intention.

A definition of an AgentSpeak(L) agent consists basically of the agent's initial beliefs (and usually the initial goals) and the agent's plans [35]. If we need to develop an agent that wants to book a concert ticket for example, we need an AgentSpeak(L) code similar to the one shown in Listing 1.

```
 1  /* Initial  Beliefs */
 2  likes(radiohead).
 3  phone_number(covo,"05112345").
 4
 5  /* Belief addition */
 6  +concert(Artist, Date, Venue) : likes(Artist) <-
 7         !book_tickets(Artist, Date, Venue).
 8
 9  /* Plan to book tickets */
10  +!book_tickets(A,D,V) : not busy(phone) <-
11         ?phone_number(V,N); /* Test Goal to Retrieve a Belief */
12         !call(N);
13         ...
14         !choose_seats(A,D,V).
```

Listing 1: AgentSpeak(L) code example, based on [5].

The Lines 2 and 3 are the initial agent's beliefs. In the Line 6, we define that a plan `!book_tickets(Artist, Date, Venue)` must be executed if a new belief `+concert(Artist, Date, Venue)` is added. But the plan will only be executed if it achieves its condition `likes(Artist)`. Between Line 9 and Line 14 we have a plan. To be executed, this plan must meet its context: `not busy(phone)`. The first action of the plan is to check if the agent has a belief: `?phone_number(V,N)`. The second action depends of the first. So, if the first action fails, the agent will not execute the second `!call(N)` and the third one `!choose seats(A,D,V)`. Next section will present an extension of AgentSpeak(L): Jason.

5.2.3 Jason

Jason is a particular implementation of AgentSpeak(L) in Java. It extends the AgentSpeak(L) language on a set of features, such as [4]: handling of plan failures, support for multi-agent system distributed over a network, a library of internal actions and an IDE in the form of a jEdit or Eclipse plugin.

Jason language syntax follows the AgentSpeak(L) syntax pattern. For example, in order to develop an agent which simple prints "hello word", we need the code shown in Listing 2 [5]:

```
start.
+start : true <- .print("Hello World!").
```

Listing 2: Hello World in Jason.

The first line of this file defines the initial belief for the agent. In this example, the agent has one single belief: `start`. The second line defines a plan for the agent. In this example, the agent has just one single plan, divided in three parts. The first part of the plan is the triggering event: `+start`. The second part is the context. It is the condition that must be fulfilled for the plan to be executed. In this example, the context will always be fulfilled (true), optionally it could be purged: `+start <- .print("Hello World!")`. The third part of the plan is its body: `.print("Hello World!")`. Although `.print("Hello World!")` looks like a belief, it is in fact an *internal action*. Jason's internal actions begin with a full stop in order to give the reader some syntactic clue and do not change the environment [5]. Examples of internal actions are `.send` and `.broadcast`, commonly used for agent communication. At the end of each line, we have a dot signal "." which is a syntactic separator. This separator is very similar with the semicolon for Java or C.

In summary, in this example, when the agent starts running, it will have the belief "start". And there is a plan which establish that when the agent acquire ("+" symbol) the belief "start", it will print the message "Hello World!". Consequently, this plan is triggered when the agent starts executing. The effect of this plan is just to show the text "Hello World!" on the console.

Now we would like to present a more complex example of Jason code that makes use of recursion: computing the factorial (specifically, the factorial of 5).

```
 1 !print_fact(5).
 2 
 3 +!print_fact(N)
 4   <- !fact(N,F);
 5      .print("Factorial of ", N, " is ", F).
 6 
 7 +!fact(N,1) : N == 0.
 8 
 9 +!fact(N,F) : N > 0
10   <- !fact(N-1,F1);
11      F = F1 * N.
```

Listing 3: Example of factorial in Jason [5].

The first line of the code shown in Listing 3 describes the agent's initial goal. The "!" signal defines that the agent has the intention to achieve the goal: prints the factorial of 5. This Jason code has three plans. The first plan (Line 3) starts adding a new goal: "!fact(N,F)" (Line 4) and when this goal is achieved, the plan prints out the factorial of N. The other two plans are both for calculating the factorial. They will be triggered when the agent acquire the goal "!fact(N,F)" (Line 7 and Line 9). The first plan establish that when the agent acquires the goal of calculate the factorial of 0, nothing must be done and the result is "1". The second plan defines that when the agent acquires the goal of calculate the factorial of N, and N is bigger than 0, then calculates the factorial of N-1, the result of this must be multiplied by N.

As we can see in Listing 3, within a plan's body, there may be a goal. When a goal is found in a plan body, the interpreter looks for a plan that achieves this goal. The agent may have a set of such plans, and these plans may or may not satisfy the required context. Wherefore, in different contexts the same invocation may result in different plans being run. Jason has much more features that are considered out of the scope of this work and more information about Jason can be found on *"Programming multi-agent systems in AgentSpeak using Jason"* [5]. The next sections present a brief introduction of other agent programming languages commonly used to the development of rational agents.

5.3 Other Agent Programming Languages

5.3.1 3APL and 2APL

Artificial Autonomous Agents Programming Language (3APL) is a combination of imperative and logic programming, in addition to agent oriented features. All regular programming constructs (such as recursive procedures and state-based computation) from the imperative programming are supported and from the logic programming, 3APL inherits the computational model for querying the belief base of an agent [21]. 3APL was created to support agent programming mechanisms, such as [21]: representing and querying the agent's beliefs; belief updating; and goal updating;

A Practical Agent Programming Language (2APL) is a BDI-based agent-oriented programming language that extends and modifies the original version of 3APL. The most important modification is the support of multi-agent systems [57]. 2APL promotes the integration between declarative and imperative style programming: the agent's beliefs and goals are implemented in a declarative programming style, while plans, events and environments are implemented in an imperative programming style.

According to Ziafati [57], the declarative programming part supports the implementation of reasoning and update mechanisms that are needed to allow individual agents to reason about and update their mental states. The imperative programming part facilitates the implementation of plans, flow of control, and mechanisms such as procedure call, recursion, and interfacing with existing imperative programming languages.

2APL separates the multi-agent and individual agent concerns, providing two different sets of programming constructs for each of them. This separation allows the development of a wide range of social organizational concepts in a modular way [57]. At the multi-agent level, 2APL creates individual agents, assigns unique names to them, creates external environments and defines the access relations between the external environments and the agents. Environments and agents are implemented as Java objects, allowing the possibility to use these objects as interfaces to the physical environments or other software. At the individual agent level, 2APL implements beliefs, goals, actions, plans, events, and three different types of rules. The first type of rule generates plans for achieving goals; the second type is responsible for processing events and handling received messages; and the last one is responsible for handling and repairing failed plans.

5.3.2 Jack Agent Language

Jack Agent Language (JAL) [7] is not a logic-based language, but an extension of Java which implements features of logic languages (such as logical variables). Jack programs are compiled as normal Java files with a pre-compiler and can subsequently be translated to Java classes using the normal Java compiler [3]. Syntactic constructs was added to Java, allowing developers to implement BDI concepts such as beliefs, goals, plans, and events [57]. Plans can be composed of reasoning methods and grouped into capabilities which, together, compose a specific ability an agent is supposed to have, thus supporting a good degree of modularization [3].

According to Lucas et al. [7], Jack provides a high performance, light-weight implementation of the BDI architecture, and can be easily extended to support different agent models or specific application requirements. Jack was designed to achieve the following goals [7]: provide to the developers robust, stable and light-weight framework; make easy the technology transfer from research to industry; satisfy practical agent application needs; and enable further applied research on agent programming.

5.3.3 Jade and Jadex

JADE (Java Agent DEvelopment Framework) is a software framework to develop agent applications in compliance with the FIPA specifications for inter-operable intelligent multi-agent systems [1]. The Foundation for Intelligent Physical Agents (FIPA) is an international non-profit association of companies and organizations sharing the effort to produce specifications of generic agent technologies [1]. Its goal is to simplify the development of multi-agent systems while ensuring standard compliance through a comprehensive set of system services and agents. It can be adapted to be used on devices with limited resources such as PDAs and mobile phones [3], and provides a set of services and graphical tools for debugging and testing. Jade deals with all those aspects that are not peculiar of the agent internals and that are independent of the applications, such as message transport, encoding and parsing, or agent life-cycle [1]. The software architecture is based on the coexistence of several Java Virtual Machines (JVM) and communication relies on Java RMI (Remote Method Invocation) between different JVMs and event signaling within a single JVM. Each JVM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host [1].

Jadex [25] is a software framework for the creation of goal-oriented agents following the belief-desire-intention (BDI) model. The framework is realized as a rational agent layer that sits on top of JADE, and supports agent development with Java and XML [3]. The Jadex reasoning engine addresses traditional limitations of BDI systems by introducing new concepts such as explicit goals and goal deliberation mechanisms. Jadex does not define a new agent programming language, but uses a BDI meta-model defined in XML-schema for agent definition and pure Java as implementation language for plans avoiding the need for a pre-compiler [25].

5.4 Multi-Agent Environments

As present in Chapter 4, agents are situated in an environment and they interact with this environment through the use of sensors and actuators. In many multi-agent applications, the environment is the real world and when an agent decides to act, the action itself is executed by a given hardware that performs changes in the environment. In the same way, perception of the environment is gathered from sensors that capture aspects of the current state of the real world. However, in other cases, agents are situated in a simulated world. In this case, in order to create a multi-agent system, not only the agents need to be developed, but also the environment must be implemented. The developer should create a computational model of the real world (or a computational model of a given artificial environment) and this computational model needs to be able to simulate the dynamic aspects of the original environment. Based on the artificial environment, the developer implements the agents that will "live" on it, acting upon it and perceiving properties of it [5]. The implementation of a simulated environment is useful even for applications that are aimed at deployment in real-world environments, and most developers will want to have a computational model of the real-world where they can verify and validate it, in addition to evaluate how well the system can perform under specific circumstances of the original environment [5].

The growing research in MAS has led the creation of several models, architectures and frameworks that help the development of environments, but present all of them is beyond the scope of this work. Among them, the CArtAgO [39] framework introduced a general-purpose computational and programming model for the development of environments based on the notion of *artifacts* [38].

Artifacts are runtime objects that provide some kind of function or service which agents can use in order to achieve their objectives. Artifacts have a set of operating instructions and a function description that can be read by the agents in order to discover its characteristics [40]. Agents are able to interact with the artifacts by invoking operations and then observing the events generated from them. Figure 5.2 shows the invoking operations as a set of buttons that agents can press to interact with the artifact and the output area which contains the observable events perceived through the sensors.

CArtAgO (Common ARTifact infrastructure for AGents Open environments) is a framework used for engineering multi-agent applications, providing an architecture based on artifact notion and all the related concepts, such as object-oriented and service-oriented abstractions, autonomous

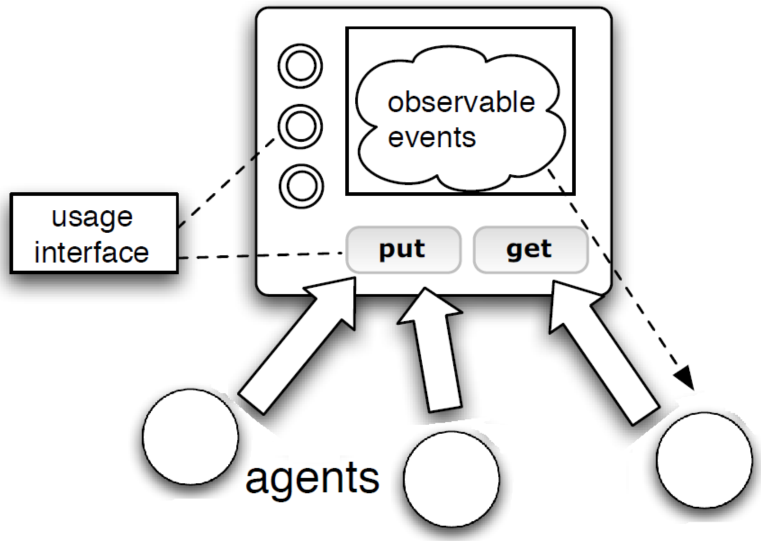

Figure 5.2 – An abstract representation of an artifact [40].

activities (typically goal / task oriented) and structures (typically passive and reactive entities which are constructed, shared and used by the agents) [39]. In CArtAgO, sensors and actuators are structures that agents can create and use to partition and to control the information flow perceived from *artifacts*. Namely, CArtAgO provide a natural way to model object-oriented and service-oriented abstractions (objects, components, services) at the agent level of abstraction [39] through the use of artifacts. We present a CArtAgO artifact pseudo-code example in Section 6.2.

5.5 Considerations

The increasing amount of research on multi-agent systems resulted in the development of various programming languages and tools that are adequate for the implementation of MAS. Analyzing the literature, several APLs stand out, such as the ones presented in the current Chapter.

The number of available APLs is a sign that MAS is becoming widely used, and that more agent-based applications may be implemented in the near future. An agent programming language should include, at least, some structure corresponding to an agent and may also have primitives such as beliefs, goals, or other mentalistic notions [56].

Using agent programming languages rather than traditional ones, proves to be useful when we need to solve a problem that is modeled in an agent-oriented fashion way, using for example goals to reach and beliefs about the state of the world; this kind of problems are very common when developing for Autonomous Mobile Robots.

6. JACAROS: THE INTEGRATION OF JASON AND ROS

The main objective of our work is to simplify the modeling of complex behaviors in robots, by using the abstraction of autonomous cognitive agents, allowing the development of mobile robots at a higher level of abstraction, following the BDI model. To fulfill this objective we created JaCaROS, an integration between an agent programming language (APL) with a robot development framework (RDF). This integration has two requirements: a *functionality requirement* and an *extensibility requirement*.

To satisfy the *functionality requirement*, we need to design and implement a complete architecture, including an interface responsible for translating APL higher-level abstraction into low-level robot control directives and translating data gathered from the environment into information that the agent can understand. To satisfy the *extensibility requirement*, our integration must support the development of new applications for mobile robots, including the design and use of new sensors and actuators supported by the RDF. In the next sections, we detail the requirements of our integration and its architecture.

6.1 Integration Requirements

To fulfill the functionality requirement, we need a *middleware* that communicates with robot hardware. This middleware must meet a set of basic requirements, including: supporting a large number of sensors, actuators and robot kits; having an active developers community; having a good documentation; and to be frequently updated. Given these requirements, ROS proved to be the best choice among the frameworks previously studied. Actually, ROS is more than a middleware, as shown in Section 3.1 and it is free and open source, providing us with a large variety of tools already implemented; it has a good technical support from community; it allows code access from the core of ROS to external modules made by other researchers; and it supports cross-language development and it supports a great number of sensors and actuators. Finally, the authors of ROS are continuously working on Player[1] source code in order to allow developers to gain access and reuse code (drivers, algorithms and simulation environments) previously implemented by Player developers within ROS platform. Player project has two simulators: Stage and Gazebo; both are used by our project and are presented in Section APPENDIX D.

To fulfill the extensibility requirement, we need an agent programming language that supports the BDI architecture; multi-agent systems; have an active developers community; and have a good documentation. Given these requirements, Jason proved to be the best choice among the previously studied APLs. As shown in Section 5.2.3, Jason is an extension of the AgentSpeak(L), which has all the most important elements of the BDI architecture; it supports strong negation, making

[1]More information about Player is presented in Section 3.5

available both closed-world and open-world assumptions; and it supports additional information in beliefs and plans.

Throughout our work, it was noticed the need of raise the abstraction level of the *extensibility requirement* (in the way agents communicate with the robot framework) and the need of a better control of data flow between the layers of our architecture. Thus, we need an abstract representation of the connection between the high level agent perception and the low level sensor perception and this was reached through the use of artifacts[2]. An infrastructure which supplies basic services for agents to instantiate and use artifacts, as well as a flexible way for multi-agent systems developers to design and construct any kind of artifact may be provided by CArtAgO[3] and it is used in our work as the main abstraction to support our intermediate architecture. Rational agent developers commonly use Jason and CArtAgO together, composing an agent-oriented programming platform knew as *JaCa*. Programmers use Jason as programming language to develop and execute the agents and they use CArtAgO as the framework to develop and execute the environments [42].

6.2 Integration Architecture

Figure 6.1 illustrates the design of our architecture which consists of three layers: the *agent layer*, in which we develop rational agents using Jason in order to interact with the robot using the BDI model; the *interface layer*, in which we establish the communication between the APL and the RDF and we use special artifacts to abstract robots sensors and actuators. We call these special artifacts *JaCaROS artifacts*; and finally, the *control layer*, in which we have access to robot's sensors and actuators in the hardware level through the use of *rosjava*[4] library.

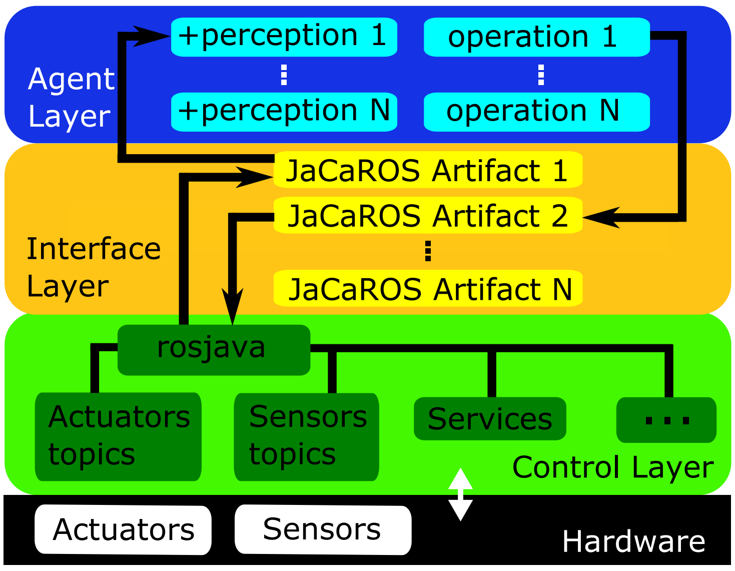

Figure 6.1 – The design of our architecture.

[2]More information about artifacts can be found in Section 5.4
[3]More information about CArtAgO is presented in Section 5.4
[4]The rosjava library is an implementation of ROS in pure-Java with Android support [47].

JaCaROS artifacts (that we use at the *interface layer*) are objects that provide the high level of abstraction we want in order to access robot sensors, robot actuators and robot control directives. Through JaCaROS artifacts *operations*, the agent is able to gather information from the environment or control the robot actions. Each agent in a multi-agent system can execute one or more external actions, each external action may call one JaCaROS artifact operation, each JaCaROS artifact can have one or more operations, each operation can publish on or subscribe to one or more ROS topics and each ROS topic can handle one or more sensors or actuators. Therefore, an agent using a single JaCaROS artifact is able to interact with one ore more sensors or actuators.

Figure 6.2 illustrates the design of our architecture with examples of Jason external actions, JaCaROS artifacts and ROS topics. In this example, a laser range finding sensor publishes data from the environment on a ROS topic called */scan*, and our *Scan* artifact is able to read the ROS topic and translate this data into information that the agent is able to understand. For the agent, the information received from the JaCaROS artifact is a new belief and the format of this belief is defined by the artifact developer. Since ROS returns an array of float values at */scan* topic, we decided to program our laser finding JaCaROS artifact to return following belief: *+laserrange(scan[0], scan[laserSize*0.25], scan[laserSize*0.5], scan[laserSize*0.75], scan[laserSize-1];)*, but the artifact developer is free to use any other design (namely, since ROS provides an array of float values for */scan*, the developer can do any computation with this data before send it to the agent, setting the abstraction level of the agent belief base for this value). In order to make the robot rotate, the agent in our example executes an external action called *rotate*, this external action executes a given operation within the *Movement artifact*, and this operation publishes data on */cmd_vel_mux* topic and finally, the actuator receives data from the ROS topic.

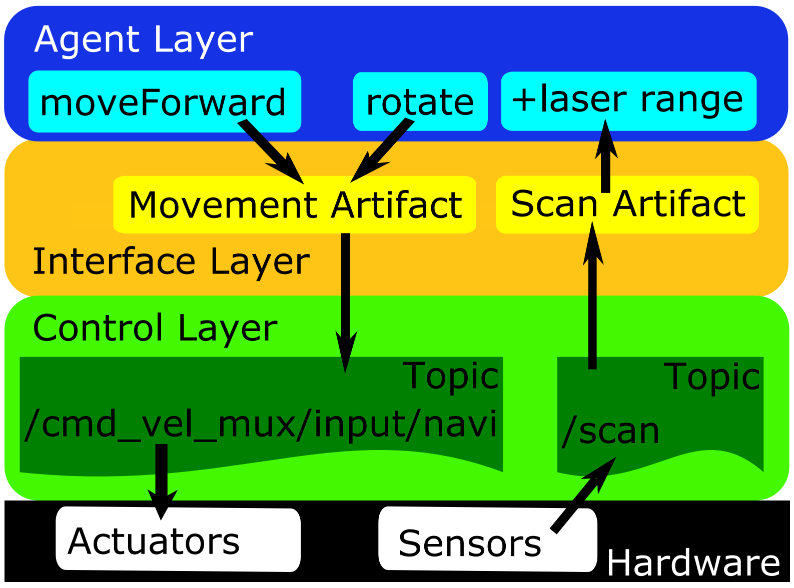

Figure 6.2 – The design of our architecture, with examples of Jason external actions, JaCaROS artifacts and ROS topics.

To better understand how the *interface layer* was works, we first briefly describe how a CArtAgO artifact is created and then we describe how a JaCaROS artifact is created. Listing 4 shows the Java pseudo-code for a CArtAgO artifact: first, to develop a new CArtAgO artifact, the developer must extend the *Artifact* class from CArtAgO library (Line 1); second, using the method *defineObsProperty* (Line 7), a developer may define an environment property to be observable by an agent and an initial value for this property; third, a set of operations (Line 11 and Line 19) can be created. These operations are used by the agent in order to interact with the environment and they may change the value of the observed properties through the use of *updateValue* method (Line 15) or read the current property value using the method *getObsProperty* (Line 13); Finally, the method *init* (Line 5) represents the artifact constructor [40] and it is executed automatically as soon as the artifact is created;

```
 1 import cartago.*;
 2 
 3 public class ClassName extends Artifact {
 4   ...
 5   void init(){
 6     ...
 7     defineObsProperty(propertyName, values)
 8     ...
 9   }
10 ...
11   @OPERATION void operationName(){
12     ...
13     ObsProperty prop = getObsProperty(propertyName);
14     ...
15     prop.updateValue(newValues);
16     ...
17   }
18 ...
19   @OPERATION void sum(){
20     ...
21   }
22 ...
23 }
```

Listing 4: CArtAgO artifact pseudo-code example.

The development of a JaCaROS artifact is a slightly different: we use *JaCaRosArtifact* class (that extends the *Artifact* class from CArtAgO) which adds features specifically designed to interact with ROS. Instead of extending the *Artifact* class from CArtAgO, our JaCaROS artifacts extend *JaCaRosArtifact* class, allowing them to interact with ROS. Thus, at a high level, when an agent observes a property exposed by a JaCaROS artifact, it is observing a message from a ROS topic and when an agent executes a JaCaROS artifact operation, it is publishing data on a ROS topic.

To start publishing on or subscribing to a ROS topic, and to start interacting with a ROS service, the JaCaROS artifact needs to establish a connection with ROS. In order to communicate with ROS topics, we need to create a node in the ROS system and connect this node with the ROS Master node[5]. Our architecture is responsible for creating a node for each artifact interested in interact with ROS topics and ROS services. We call our node within ROS system *Jason node*.

[5]More information about ROS Master node is presented in Section 3.1

Figure 6.3 shows the design of the connection between a JaCaROS artifact and ROS: basically this is done by the creation of a *Jason node* into ROS system. First, the agent creates a JaCaROS artifact (red bubble 1) after that, our architecture first, asks ROS to create a new node (red bubble 2), and ROS returns the created node (red bubble 3); and second, it asks ROS to connect the new node with ROS Master node (red bubble 4); finally, if the connection is successful established, the artifact becomes available to the agent.

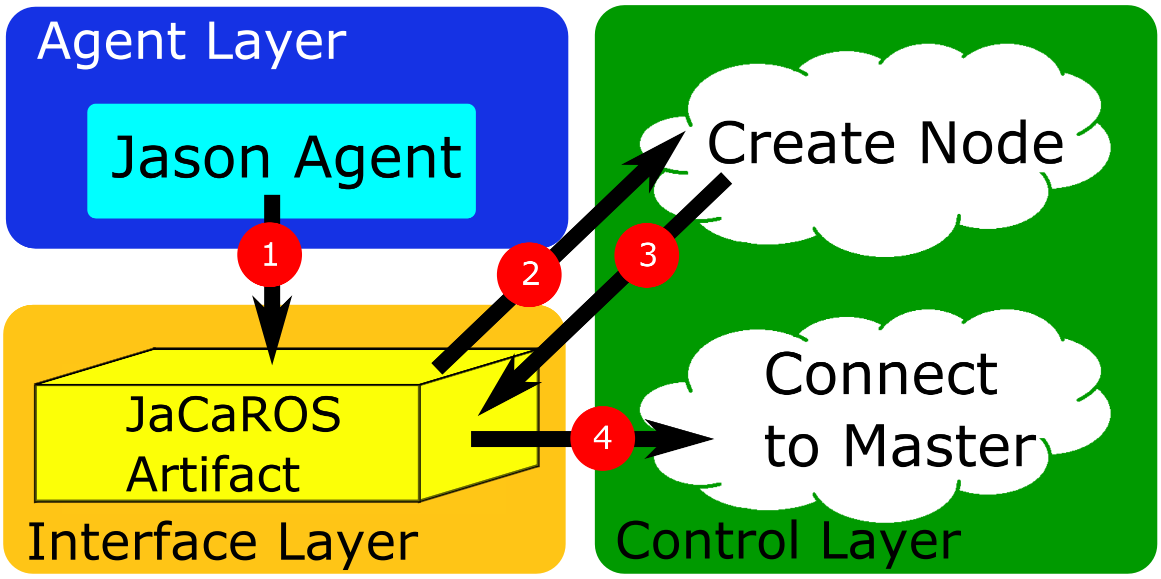

Figure 6.3 – JaCaROS artifact connection with ROS.

6.3 JaCaROS Artifacts Architecture

We developed a basic set of JaCaROS artifacts (such as artifacts for motion, odometry and laser range finding) to compose the basic API and we present four of them (ArtOdometry, ArtOdometryOnDemand, ArtTutorial and ArtCmdVelAndOdometry) in order to explain how the architecture of the JaCaROS artifacts works. Our architecture allows the creation of new artifacts, according to the Jason/ROS developer needs, thereby ensuring support for new sensors and actuators supported by ROS.

We divide them by artifacts that use actuators, artifacts that use sensors and artifacts which interact with ROS services, but in practice, a set of sensors, actuators and services can be controlled by a single JaCaROS artifact. We start by presenting a JaCaROS artifact responsible for controlling sensors and our architecture has two modes of operation for this kind of artifact: the *verbose mode* in which the artifact forwards to the agent all the information gathered by the sensor; and the *on demand mode* in which the agent must request the gathered information by using an artifact operation.

Figure 6.4 ilustrates how *verbose mode* works. We used *ArtOdometry* artifact as an example because basically, all the artifacts using the *verbose mode* works in a similar way. The

ArtOdometry is responsible for reading the odometry of a given movement and its code is presented in Section APPENDIX B. As we can see in Figure 6.4, when the sensor detects a given information from the environment (red bubble 1), ROS publish this information into the topic (red bubble 2): in our example, ROS publish in */odom* topic; A listener within the artifact receives the published information (red bubble 3) and saves the information into a local variable (red bubble 4); After that, our artifact updates the observable property (red bubble 5) and as a consequence, the information is propagated to the agent (red bubble 6).

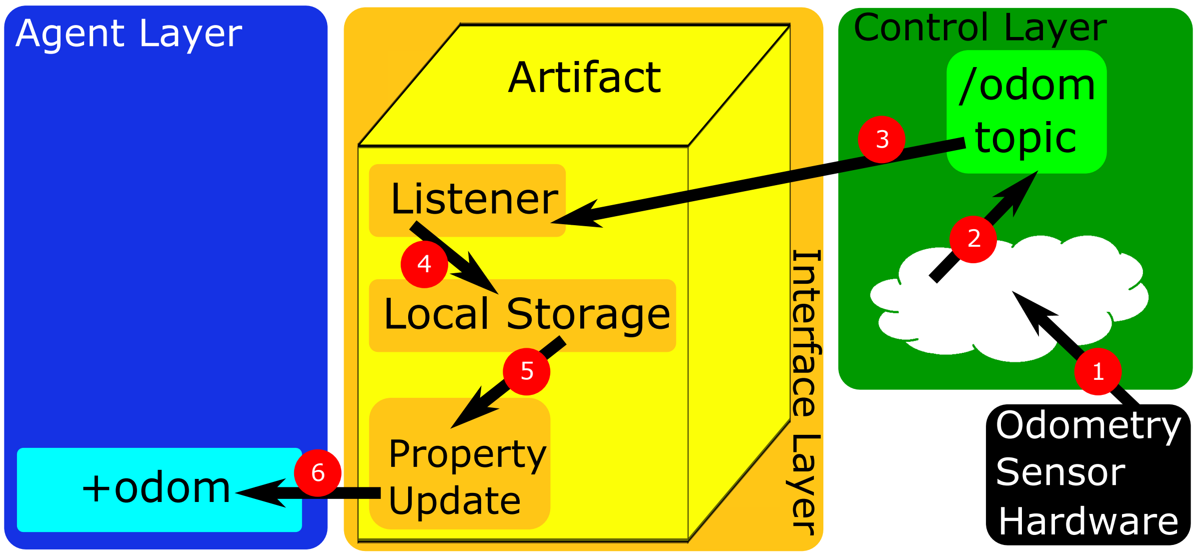

Figure 6.4 – Design of ArtOdometry artifact (verbose mode example).

We can observe that the *verbose mode* may cause an agent's overhead data processing or a high refresh rate in the agent's belief-base, since in this mode the artifact forwards to the agent all the data gathered from the sensor. In order to avoid these problems, a set of actions is taken, including:

- to avoid duplicate information over communication between the agent and the artifact, it discards the gathered information when the value of the current perception (the value published by ROS and received by the artifact) is the same of the last perception value (the value previously stored on the artifact) and the artifact does not forward the message;
- the artifacts have a refresh rate that can be adjusted, thus, some messages published into the ROS topic can be discarded. Namely, the published messages can be updated on the artifact and not forwarded to the agent, and this can occurs in a given rate.
- the artifacts can be focused and unfocused by the agent at any time, avoiding the receipt of messages in a moment that a given information is not needed.
- to avoid a large agent's belief-base, we adopted the range of view[6] concept for a given type of artifact, in which the artifact just sends to the agent information limited by a range, relative to the current agent's position.

[6]More information about the range of view is presented in Section APPENDIX B

The JaCaROS artifact developer is free to program any other behavior to avoid the high refresh rate in the agent's belief-base, including changing the methods listed above. For example: instead of discarding only identical messages (the first item of previous list), namely, instead of always forward a message that is different from the previous one, the developer can defines that a message will be forwarded only if there is a significant change in its value.

Our architecture makes available a mechanism to gather information from a given topic on demand, helping to avoid unnecessary updates on the agent's belief base. These kind of artifact does not update the observable property for each message received, but instead of it, the artifact updates a local variable responsible for storing the last value received from the ROS topic and only updates the observable property through the use of an operation. As a result, the agent's belief base is updated only after the execution of an operation. We call this *on demand mode* and Figure 6.5 illustrates how it works.

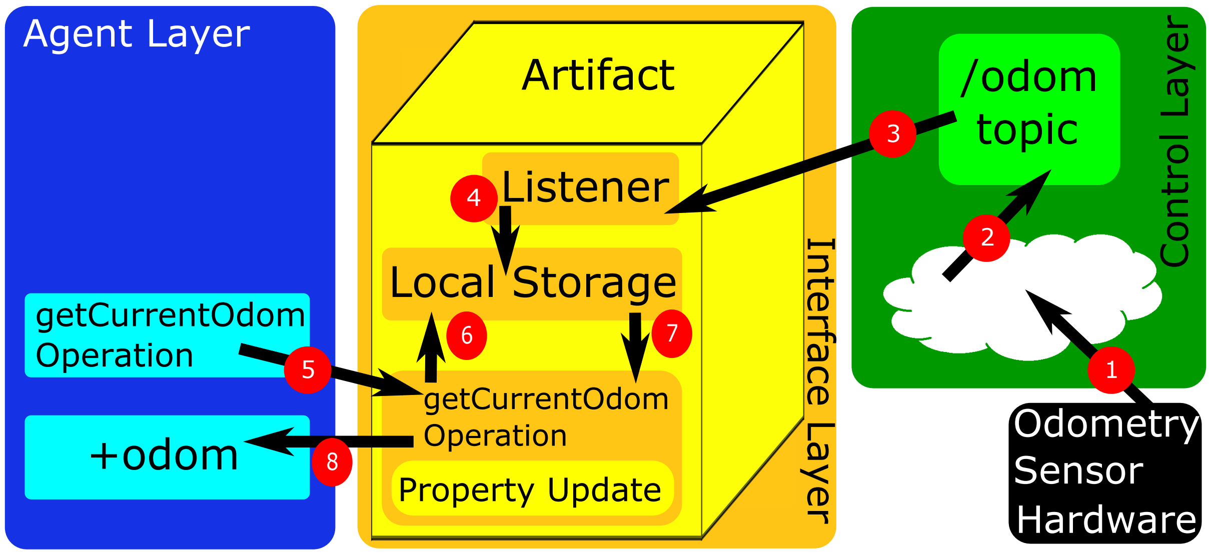

Figure 6.5 – Design of ArtOdometryOnDemand artifact (on demand mode example).

We used *ArtOdometryOnDemand* artifact as an example because basically, all the subscriber artifacts using the on demand mode work in a similar way. The ArtOdometryOnDemand is responsible for reading the odometry of a given movement and its code is presented in Section APPENDIX B. As we can see in Figure 6.5, when the sensor detects a given information from the environment (red bubble 1), ROS publishes this information into the topic (red bubble 2): in our example, ROS publishes in */odom* topic; A listener within the artifact receives the published information (red bubble 3) and saves the information into a local variable (red bubble 4); at this time, our artifact does not update the observable property, instead, it delays the update for the time that agent asks for it, through the use of an operation (red bubble 5). The artifact operation is responsible for verifying the value previously saved (red bubble 6) and use it to update the observable property (red bubble 7); the consequence is dissemination of the information to the agent (red bubble 8).

In our architecture, on demand artifacts are also used to interact with ROS services and Figure 6.6 illustrates how it works. We used *ArtTutorial* artifact as an example because basically, all the artifacts that interact with ROS services works in a similar way. In our example, first, through the use of an operation (red bubble 1), that agent asks for a given service (sum two values); after that, the JaCaROS artifact call the ROS service (red bubble 2). A listener within the artifact receives the service response (red bubble 3) and saves the information into a local variable (red bubble 4); After that, our artifact updates the observable property (red bubble 5) and as a consequence, the information is propagated to the agent (red bubble 6).

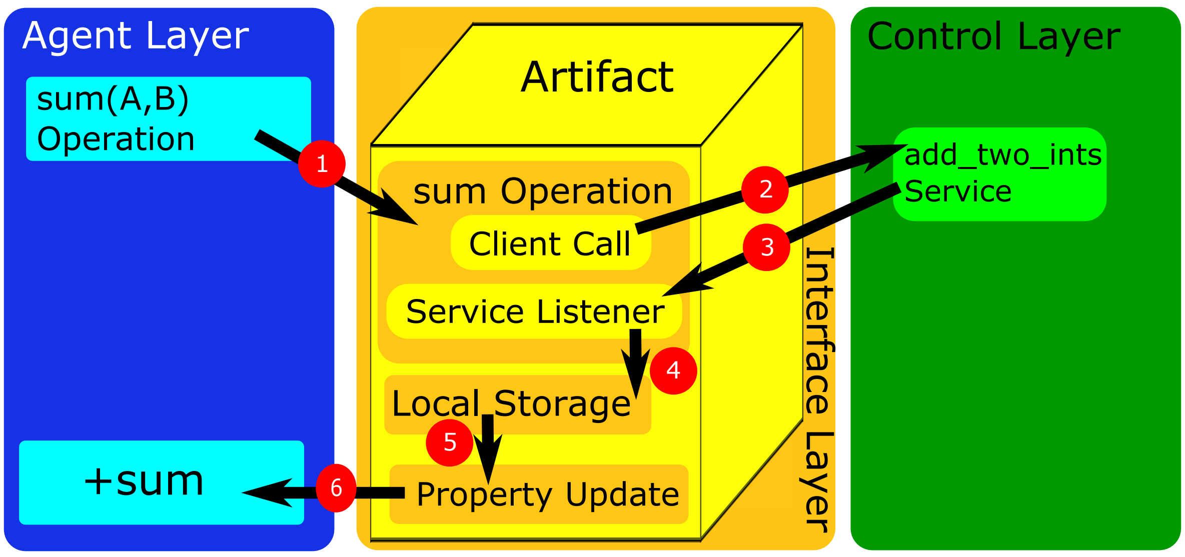

Figure 6.6 – Design of ArtTutorial artifact: an example of interaction with a ROS service.

Now that we presented how JaCaROS artifacts responsible for controlling sensors work and how JaCaROS artifacts interact with ROS services, we introduce how an artifact responsible for controlling actuators works: Figure 6.7 shows the design of our *ArtCmdVelAndOdometry* artifact. We used this artifact as an example because basically, all the artifacts that interact with ROS system using a *publisher* works in a similar way. In our example, first, the agent executes a move operation (red bubble 1); after that, the JaCaROS artifact publishes the movement information into */cmd_vel* topic (red bubble 2); and as a consequence, the information published into the topic will be transmitted to the robot actuators (red bubble 3 and red bubble 4). Meanwhile, the artifact receives information from odometry (red bubble 6) in order to control the distance already traveled (red bubble 5) and it uses this information to know when to stop publishing into the ROS topic. A second way to stop publishing in the topic is abort the move operation by using the abort operation (red bubble 7). If fact, as we can see in Figure 6.7 ArtCmdVelAndOdometry artifact controls actuators and sensors because the agent needs to know how long it traveled, in order to know when to stop moving.

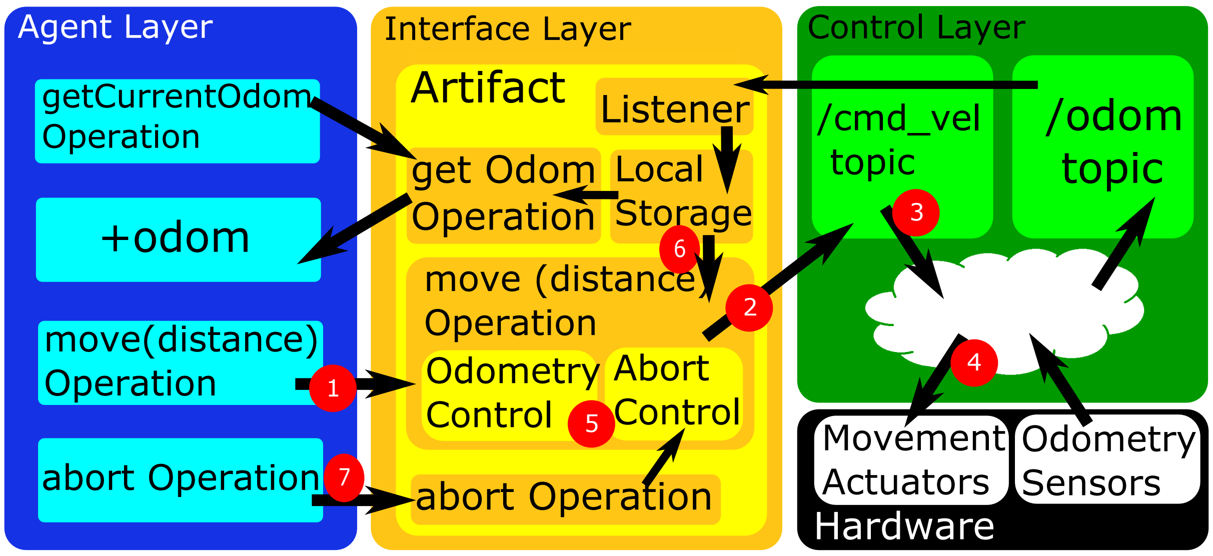

Figure 6.7 – Design of ArtCmdVelAndOdometry artifact: an example of publish artifact.

7. IMPLEMENTATION

Figure 7.1 illustrates the class diagram of JaCaROS architecture and three JaCaROS artifacts used as example. In the center of this figure, we have the *JaCaRosArtifact* class, and as presented in Section 6.2, this class has features specifically designed to interact with ROS. *JaCaRosArtifact* class has two main functions: to create a node within ROS system; and to provide means for the JaCaROS artifacts to interact with ROS topics. We explain these functions below.

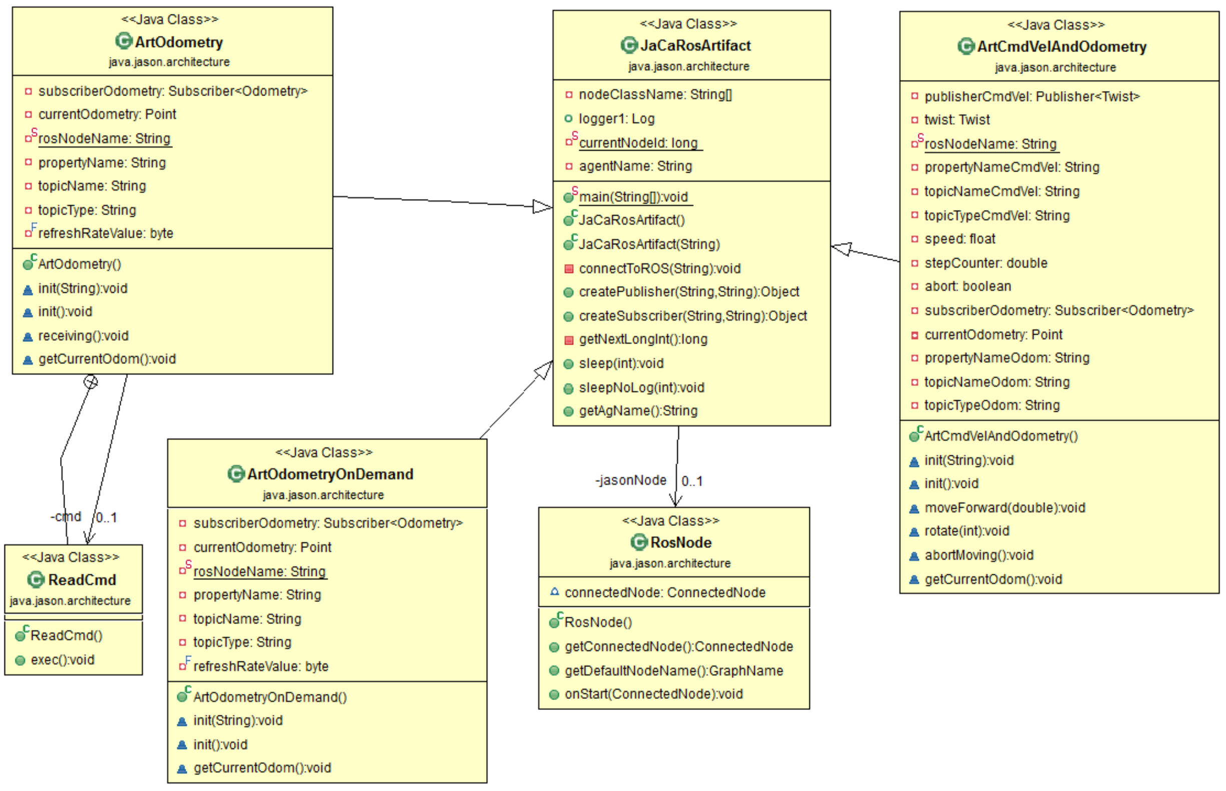

Figure 7.1 – Class Diagram of JaCaROS Architecture (including three artifacts).

The node used by JaCaROS artifacts is an instance of *RosNode* class, as we can see in Figure 7.1. Listing 5 shows the pseudo-code of the *JaCaRosArtifact* class constructor and the method responsible for asking ROS for a new node. To improve readability, we omit the least significant lines, showing only how the class constructor establishes a connection with ROS. This code snippet is called once for each JaCaROS artifact created by the agent, and it is responsible for two objectives: to create a new Jason node into ROS system; and to connect it to ROS Master node. *JaCaRosArtifact* class receives the node name from the artifact (Line 5) and calls *connectToROS* method in Line 7. This method creates a new node (Line 14) and it tries to connect the created node with the ROS Master node in Line 16. When the connection is successful established, the Jason node is ready to be used by our JaCaROS artifact.

In addition to create a node and connect it to ROS Master, the *JaCaRosArtifact* class is responsible for providing means for the JaCaROS artifact to publish on and to subscribe to a ROS topic, namely, *JaCaRosArtifact* class provides a *subscriber object* for JaCaROS artifacts which want

```java
public class JaCaRosArtifact extends Artifact {
  ...
  RosNode m_rosnode;
  ...
  public JaCaRosArtifact(String name) {
    ...
    connectToROS(name);
    ...
  }
  private void connectToROS(String name) {
    ...
    name = name + "_" + getNextLongInt(); //avoid jason nodes with the same name
    nodeConfiguration.setNodeName(name);
    RosNode m_rosnode = (RosNode)loader.loadClass(nodeClassName)
    ...
    while( m_rosnode.getConnectedNode() == null ) sleep(1000);
    ...
  }
}
```

Listing 5: JaCaRosArtifact class pseudo-code (class constructor and connectToROS method).

to subscribe to a ROS topic, and a *publisher object* for JaCaROS artifacts which want to publish on a ROS topic. We can see the methods responsible for providing these objects in the class diagram (Figure 7.1), they are called *createSubscriber* and *createPublisher*. As shown in Listing 6, in order to create a publisher, we use the method *createPublisher* (Line 3) and its parameters are the ROS topic name and the ROS message type and it returns a publisher which our JaCaROS artifact use in order to publish data into the ROS topic. To create a subscriber, we use the method *createSubscriber* (Line 7) and its parameters are the same from *createPublisher* and it returns the subscriber which is used by our JaCaROS artifacts in order to receive data from a given ROS topic.

```java
 1 public class JaCaRosArtifact extends Artifact {
 2   ...
 3   public Object createPublisher(String topicName, String topicType){
 4     Object result = m_rosnode.getConnectedNode().newPublisher(topicName, topicType);
 5     return result;
 6     }
 7   public Object createSubscriber(String topicName, String topicType){
 8     Object result = m_rosnode.getConnectedNode().newSubscriber(topicName, topicType);
 9     return result;
10   }
11   public Object createServer(String serviceName, String serviceType, ServiceResponseBuilder
        builder) {
12     Object result = m_rosnode.getConnectedNode().newServiceServer(serviceName, serviceType,
          builder);
13     return result;
14   }
15   public Object createClient(String clientName, String serviceType) throws
        ServiceNotFoundException{
16     Object result = m_rosnode.getConnectedNode().newServiceClient(clientName, serviceType);
17     return result;
18   }
19   ...
20 }
```

Listing 6: JaCaRosArtifact class pseudo-code (methods which create publishers and subscribers).

JaCaRosArtifact class is also responsible for providing means for JaCaROS artifacts to interact with a ROS service. As shown in Listing 6, in order to create a client, we use the method *createClient* (Line 15) and its parameters are the ROS service name and the ROS message type. It returns a client object which our JaCaROS artifact use in order to interacts with a given ROS service. The ROS topic/server name and the ROS message type are obtained through the use of command-line tools from ROS system, such as *rostopic* and *rosservice*. The study and presentation of these tools in details are out of the scope of this work and the interested reader can find more information in "Ros: an open-source robot operating system" [34].

The creation of Jason node and the creation of the publishers and subscribers happens when a JaCaRos artifact is created[1], as illustrated in Figure 7.2. First, when the JaCaROS artifact is created, its constructor calls *JaCaRosArtifact* class constructor. It calls *connectToRos* method which creates a Jason node within ROS system. After that, the JaCaROS artifact executes its init() method and this method calls *createSubscriber*, *createPublisher* or *createClient* method. As explained before, this method returns a subscriber or a publisher that is used by the artifact to interact with ROS topics.

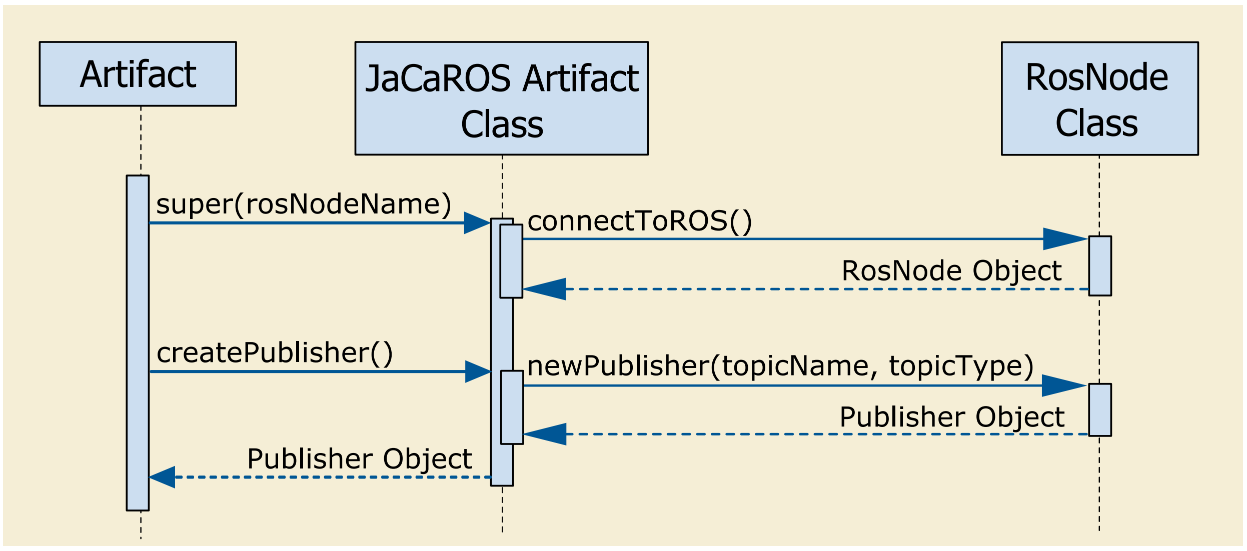

Figure 7.2 – Sequence diagram of Jason node creation and publisher creation.

7.1 Code Development Using JaCaROS

As we can see in the class diagram (Figure 7.1), all the JaCaROS artifacts are extended from *JaCaRosArtifact* class, inheriting the characteristics needed to interact with ROS. In order to create JaCaROS artifacts, a set of proceedings must be done by the agent. These proceedings are the same that an agent does in order to create and use a common CArtAgO Artifact. At these level, the changes made in order to integrate Jason and ROS are transparent to Jason developer: Listing 7 illustrates the plans needed to create and use a new JaCaROS artifact (ArtOdometry artifact is

[1]More information about how JaCaRos artifacts are created is presented in Section 7.1

used as an example). To improve readability, we omit the least significant lines (such as repairing plan) of this pseudo-code.

```
 1  +!create_odom(AgentName, OdomId) :
 2     <- .print("Creating Odometry artifact. We are using an agent name: ", AgentName);
 3        .concat("Odom", AgentName, Result);
 4        makeArtifact(Result,"jason.architecture.ArtOdometry",[AgentName],OdomId).
 5
 6  +!create_odom :
 7     <- .print("Creating Odometry artifact. The agent name is unknown.");
 8        makeArtifact("Odom","jason.architecture.ArtOdometry",[],OdomId).
 9
10  +?toolOdom(AgentName, OdomId) :
11     true <-
12         .print("Discovering Odometry artifact.");
13         .concat("Odom", AgentName, Result);
14         lookupArtifact(Result, OdomId).
15
16  +?toolOdom(OdomId) :
17     <- .print("Discovering Odometry artifact.");
18        lookupArtifact("Odom",OdomId).
19  ...
20  +odom(A,B,C)
21     <- .print("An odometry property was perceived");
22        -+currentPose(A,B,C).
```

Listing 7: Jason plans to create and use a JaCaROS artifact (ArtOdometry was used as example).

To create and use a JaCaROS artifact, the agent exploits `makeArtifact` action (Line 4 and Line 8) and `lookupArtifact` action (Line 14 and Line 18); and in order to take a given action when the agent perceives a new value from the observation property, the developer may add a triggering event to this property (Line 20). In a multi-agent system, more then one agent may have sensors and actuators with the same name. Therefore, the artifact needs to know which of them are the correct one and the disambiguation is done by using a parameter on the artifact creation, as shown in Line 1 and Line 10: the `AgentName` parameter and the concatenation in Line 3 and Line 13.

Listing 8 illustrates Jason pseudo-code that uses the set of plans previously presented in order to create and use the Odometry artifact. In Line 1 agent establishes the goal to create the artifact; In Line 3, the agent will try to discover the artifact in order to use it. At any time, the agent may select which parts (artifacts) of the environment to observe by using a *focus* action (Line 5) and the agent may want to stop receiving information from a given part of the environment by using a *stopFocus* action (Line 7). The Jason developer can use the focus / stopFocus combination to limit the amount of information received by the agent.

7.2 JaCaROS Artifacts Implementation

To present the implementation of JaCaROS artifacts, we use pseudo-code blocks of four existing artifacts as example: ArtOdometry, ArtOdometryOnDemand, ArtCmdVelAndOdometry and

```
!create_odom();
...
?toolOdom(OdomID);
...
focus(OdomID);
...
stopFocus(OdomID);
```

Listing 8: Jason pseudo-code using plans to create and use ArtOdometry artifact.

ArtTutorial. As we presented in Section 6.2, to develop a new JaCaROS artifact, we need to extend the JaCaRosArtifact class and define a set of properties within the artifact, such as: ROS topic/service name, ROS message type, Jason node name and the subscriber/publisher object.

7.2.1 Basic Properties of a ROS Artifact

Listing 9, shows the set of properties needed by the artifacts ArtOdometry and ArtOdometryOnDemand. We present them as example of artifact which uses a *subscriber object* to interact with ROS. The properties needed by all the artifacts which use a *subscriber object* are similar: in Line 3 we define the name that is used when the Jason node is created into ROS system (variable *rosNodeName*); in Line 4, we create the object that receives the *subscriber object* from JaCaRosArtifact class (in the current example, the subscriber object is called *subscriberOdometry*). The ROS topic name is defined in the property *topicName* (Line 5) and the topic's message type is defined in the property *topicType* (Line 6); Line 7 defines the name of the property to be observable by the artifact (variable *propertyName*), it is used as parameter on CArtAgO primitives such as *defineObsProperty* and *getObsProperty*; the constant *refreshRateValue* is the artifact refresh update rate (Line 11); *init* method (Line 13) is responsible for creating the subscriber (Line 15) that is used to receive data from ROS topic. To improve readability, we omit the least significant lines of this pseudo-code.

Listing 10, shows the set of properties needed by the artifact ArtCmdVelAndOdometry. We present it as example of artifact which uses a *publisher object* to interact with ROS. The properties needed by all the artifacts which use a *publisher object* are similar: the name used when Jason node is created into ROS system is set in Line 3; the publisher responsible for publishing data into the ROS topic is defined in Line 4 (and its name is *publisherCmdVel*); in Line 5, the ROS topic name is defined; and the ROS topic type is set in Line 6. As this example is about a movement artifact, some specific properties must be defined, such as: the movement speed value to be published on the ROS topic (*speed* variable in Line 8); a variable that controls the distance traveled by the robot (*stepCounter* in Line 10); a flag to control movement interruption (*abort* flag in Line 11); and a variable of type *geometry_msgs.Twist* used to push data movement into ROS topic (Line 9). As Listing 10 illustrates, *init* method (Line 13) is responsible for creating the publisher (Line 15) that is used to publish data into ROS topic. To improve readability, we omit the least significant lines of this pseudo-code.

```
public class ArtOdometry extends JaCaRosArtifact {
  /*Basic properties*/
  private static String rosNodeName = "ArtOdometry";
  private Subscriber<nav_msgs.Odometry> subscriberOdometry;
  private String topicName = "/odom";
  private String topicType = nav_msgs.Odometry._TYPE;
  private String propertyName = "odom";
  /* Specific properties */
  private Point currentOdometry;
  private final byte refreshRateValue = 1; //must be > 0
  private byte refreshRateAux = refreshRateValue;
  ...
  void init(String agentName) {
    ...
    subscriberOdometry = (Subscriber<Twist>) createSubscriber(topicNameCmdVel, topicTypeCmdVel);
    ...
    }
  ...
  }
```

Listing 9: Pseudo-code used by ArtOdometry and ArtOdometryOnDemand artifacts: setting basic properties of an artifact which uses a subscriber.

```
public class ArtCmdVelAndOdometry extends JaCaRosArtifact {
  /*Basic properties*/
  private static String rosNodeName = "ArtCmdVelAndOdometry";
  private Publisher<geometry_msgs.Twist> publisherCmdVel;
  private String topicName = "/cmd_vel";
  private String topicType = geometry_msgs.Twist._TYPE;
  /* Specific properties */
  private float speed=1;
  private geometry_msgs.Twist twist;
  private double stepCounter = 0;
  private boolean abort = false;
  ...
  void init(String agentName) {
    ...
    publisherCmdVel = (Publisher<Twist>) createPublisher(topicNameCmdVel, topicTypeCmdVel);
    ...
    }
  ...
  }
```

Listing 10: ArtCmdVelAndOdometry artifact pseudo code: setting basic properties of an artifact which uses a publisher.

Listing 11, shows the set of properties needed by the artifact *ArtTutorial*. This is a example of artifact that interacts with ROS service. The basic properties needed by all these *client artifacts* are similar: in Line 3 we define the name that is used when the Jason node is created within the ROS system (variable *rosNodeName*); in Line 4, we create the client object that is responsible to send and receive messages of a ROS service (in the current example, the client object is called *serviceClient*). The ROS service name is defined in the property *serverName* (Line 5) and the service's message type is defined in the property *serverType* (Line 6); Line 7 defines the name of the property to be observable by the artifact (variable *propertyName*), it is used as parameter on CArtAgO primitives such as *defineObsProperty* and *getObsProperty*; *init* method (Line 11) is responsible for creating the client (Line 13). To improve readability, we omit some lines of code that are irrelevant for the definition of properties and which we explain in the following sections.

```
1   public class ArtTutorial extends JaCaRosArtifact {
2     /*Basic properties */
3     private static String rosNodeName = "ArtTutorial";
4     private ServiceClient<AddTwoIntsRequest, AddTwoIntsResponse> serviceClient;
5     private serverName = "add_two_ints";
6     private serverType = AddTwoInts._TYPE;
7     private String propertyNameSum = "twoIntsSum";
8     /* Specific properties */
9     private long currentSum;
10    ...
11    void init(String agentName) {
12      ...
13      serviceClient = (ServiceClient<AddTwoIntsRequest, AddTwoIntsResponse>) createClient(
          serverName, serverType);
14      ...
15      }
16    }
```

Listing 11: ArtTutorial artifact pseudo code: setting basic properties of an artifact which interacts with a ROS service.

7.2.2 Subscriber Artifact - Verbose Mode

Now we want to present how a *subscriber* JaCaROS artifact which uses the *verbose mode* is developed. We used ArtOdometry artifact as an example because basically, all the artifacts using the verbose mode work in a similar way. Listing 12 shows the *init* method of the ArtOdometry artifact and, as explained before, this method is executed automatically as soon as the artifact is created. In our architecture, the *init* method executes right after the artifact constructor method, namely, *init* method executes after the Jason node has a connection established with the ROS Master node.

As illustrated in Listing 12, Line 6, the artifact defines the name of the property to be observable and its initial value; In Line 8, the artifact updates the topic name with the received parameter in order to make the ROS topic disambiguation in a multi-agent environment; In Line 10, the artifact calls *createSubscriber* method in order to create a new subscriber and in Line 11, a listener is added on the subscriber thus any information from the topic can be handled by the artifact. In our ArtOdometry example (Listing 12), when the artifact receives a new odometry message (Line 13), it gathers the position information from the message (Line 14) and if the received values are different from the current values saved on the artifact (Line 15), then the current values are updated (Line 17). Actually, as shown in Listing 12 Line 15, before update local values, the artifact also checks the refresh rate defined by the Jason developer (Listing 9, Line 10) and we present the code that controls the update refresh rate in Listing 13.

```
 1  void init() {
 2    init(null);
 3  }
 4  ...
 5  void init(String agentName) {
 6    defineObsProperty(propertyName, 0,0,0); //Define Property and Initialize values
 7    if (agentName != null)
 8      topicName = "/" + agentName + topicName; // Update topic name with agent name
 9    cmd = new ReadCmd();
10    subscriberOdometry = (Subscriber<nav_msgs.Odometry>) createSubscriber(topicName, topicType
        );
11    subscriberOdometry.addMessageListener(new MessageListener<nav_msgs.Odometry> () {
12      @Override
13      public void onNewMessage(nav_msgs.Odometry message) {
14        Point localPose = message.getPose().getPose().getPosition();
15        if ( !currentOdometry.equals(localPose) && refreshRate() )
16          { // Dont update if (current = previous) && update only at correct  refresh  rate
17          currentOdometry = localPose;
18          execInternalOp("receiving"); // receiving calls await, that  calls  prop update
19          }
20        }
21    });
22  }
```

Listing 12: ArtOdometry artifact pseudo-code: init methods of verbose mode.

```
subscriberOdometry.addMessageListener(new MessageListener <nav_msgs.Odometry> () {
  @Override
  public void onNewMessage(nav_msgs.Odometry message) {
    ...
  }
  private boolean refreshRate() {
    boolean result = false;
    if (refreshRateAux % refreshRateValue == 0)
    {
      refreshRateAux = refreshRateValue;
      result = true;
    }
    refreshRateAux++;
    return result;
  }
});
```

Listing 13: ArtOdometry artifact pseudo code: developer can choose update refresh rate.

As Listing 12 illustrates in Line 11, in order to gather information from the ROS topic, JaCaROS artifacts adopt a design technique knew as Observer Pattern[2], registering listeners on the subscribers. This implies inversion of control, namely, onNewMessage (Line 13) is executed by a ROS thread, not by a CArtAgO thread, and this may generate interference in the CArtAgO layer, especially if the method executed by a ROS thread needs to update the observed property values or signal it. Fortunately, CArtAgO provides a mechanism to suspend the execution of an artifact operation until a specified command has been executed [40] and this is done by the *await* primitive which accepts an object of type *IBlockingCommand* representing a command to be executed (Listing 14, Line 7).

[2]More information about design patterns can be found in the book Design Patterns: Elements of Reusable Object-Oriented Software [13].

```
1  ...
2  @INTERNAL_OPERATION
3  void receiving() {
4    await(cmd);
5  }
6  /* The ReadCmd implements a blocking command – implementing the IBlockingCmd interface – containing the
       command code in the exec method. */
7  class ReadCmd implements IBlockingCmd {
8    public void exec() {
9      try {
10       ObsProperty prop = getObsProperty(propertyName);
11       prop.updateValues(currentOdometry.getX(),currentOdometry.getY(),currentOdometry.getZ()
           );
12       }
13     catch (Exception ex) {
14       ex.printStackTrace();
15     }
16   }
17 }
18 ...
```

Listing 14: ArtOdometry artifact pseudo-code: blocking commands.

When the artifact receives a message from the ROS topic, it should update the observed property and in our architecture this is done by using the *await* primitive protection, called within the *receiving* internal operation, as Listing 14 illustrates in Line 4. Namely, the artifact calls the *receiving* internal operation within *onNewMessage* method (Listing 12, Line 18), then this internal operation calls the *await* primitive (Listing 14, Line 4) that suspends the artifact operation until the observed property update gets completed (within *exec* method), as shown in Listing 14, Line 8).

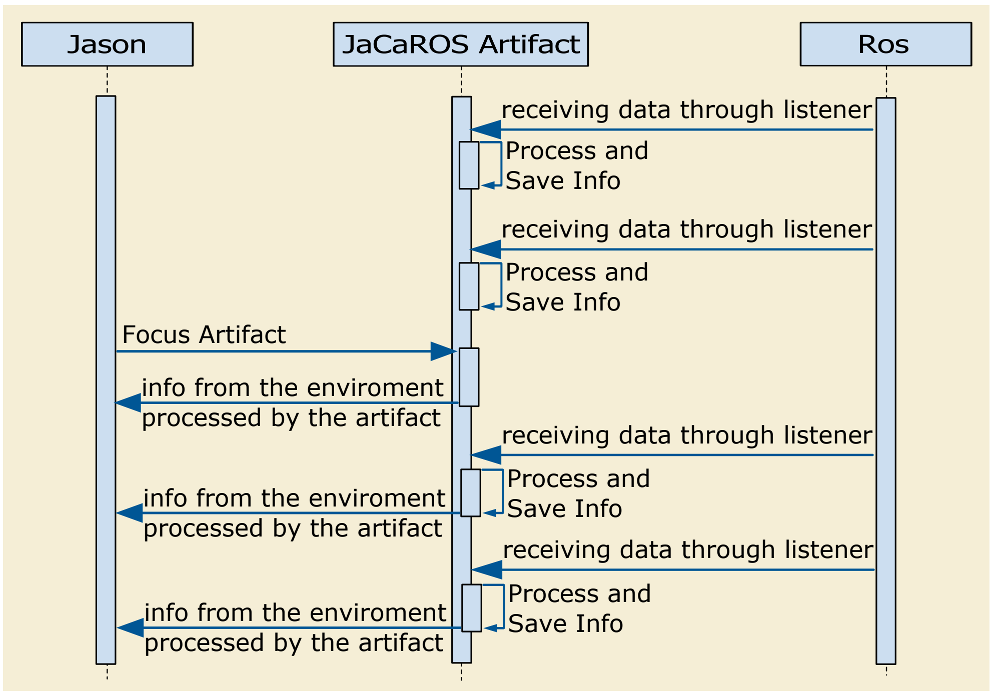
Jason
JaCaROS Artifact
Ros
receiving data through listener
Process and Save Info
receiving data through listener
Process and Save Info
Focus Artifact
info from the enviroment processed by the artifact
receiving data through listener
Process and Save Info
info from the enviroment processed by the artifact
receiving data through listener
Process and Save Info
info from the enviroment processed by the artifact

Figure 7.3 – Sequence diagram of a perception from the environment, verbose mode.

Figure 7.3 presents the processes sequence that occurs when a JaCaROS artifact - which uses verbose mode - perceives a new information from the ROS topic. The artifact is always listening to the ROS topic and receiving data from it. The artifact processes the received data and saves it locally. This process repeats for each message published on the ROS topic. At any time, an agent can focus the JaCaROS artifact and, from that moment, the agent starts receiving the perceptions generated by the artifact. At any time, the agent can stop focusing the JaCaROS artifact and, as a consequence, it stops receiving the perceptions generated by the artifact.

7.2.3 Subscriber Artifact - On Demand Mode

This section presents how a *subscriber* JaCaROS artifact which uses the *on demand* mode is developed. We used ArtOdometryOnDemand artifact as an example because basically, all the artifacts using the on demand mode work in a similar way. As shown in Listing 15, *onNewMessage* method (Line 6) does not have a call for an internal operation responsible for updating the observable property. Instead, the property update is done by an artifact operation (Line 14). Namely, we delay the observable property update, moving it from the *onNewMessage* method to an operation (in our example, the operation is called *getCurrentOdom*) and the agent can use this operation any time it needs data from the ROS topic.

```
 1  ...
 2  void init(String agentName) {
 3    ...
 4    subscriberOdometry.addMessageListener(new MessageListener<nav_msgs.Odometry> () {
 5      @Override
 6      public void onNewMessage(nav_msgs.Odometry message) {
 7        Point localPose = message.getPose().getPose().getPosition();
 8        if (!currentOdometry.equals(localPose) && refreshRate() )
 9          currentOdometry = localPose;
10        }
11    });
12  }
13  ...
14  @OPERATION void getCurrentOdom() {
15    ObsProperty prop = getObsProperty(propertyName);
16    if(currentOdometry!= null)
17      prop.updateValues(currentOdometry.getX(),currentOdometry.getY(),currentOdometry.getZ());
18    else
19      prop.updateValues(0,0,0);
20
21    signal(propertyName);
22  }
```

Listing 15: ArtOdometryOnDemand artifact pseudo code: init method of on demand mode. Agent gathers information using getCurrentOdom operation.

Figure 7.4 presents the processes sequence that occurs when a JaCaROS artifact - which uses on demand mode - perceives a new information from the ROS topic: the artifact receives a message published on the ROS topic and saves it locally, in the same manner as the previous working mode. Any new message published on the ROS topic is updated within the JaCaROS artifact; When an agent focus the artifact, it becomes eligible to receive messages published on the ROS topic. At this point, an initial update is sent to the agent, containing the current value saved from the observable property, but the agent only receives (more) information from the artifact if the agent directly request it by using an operation.

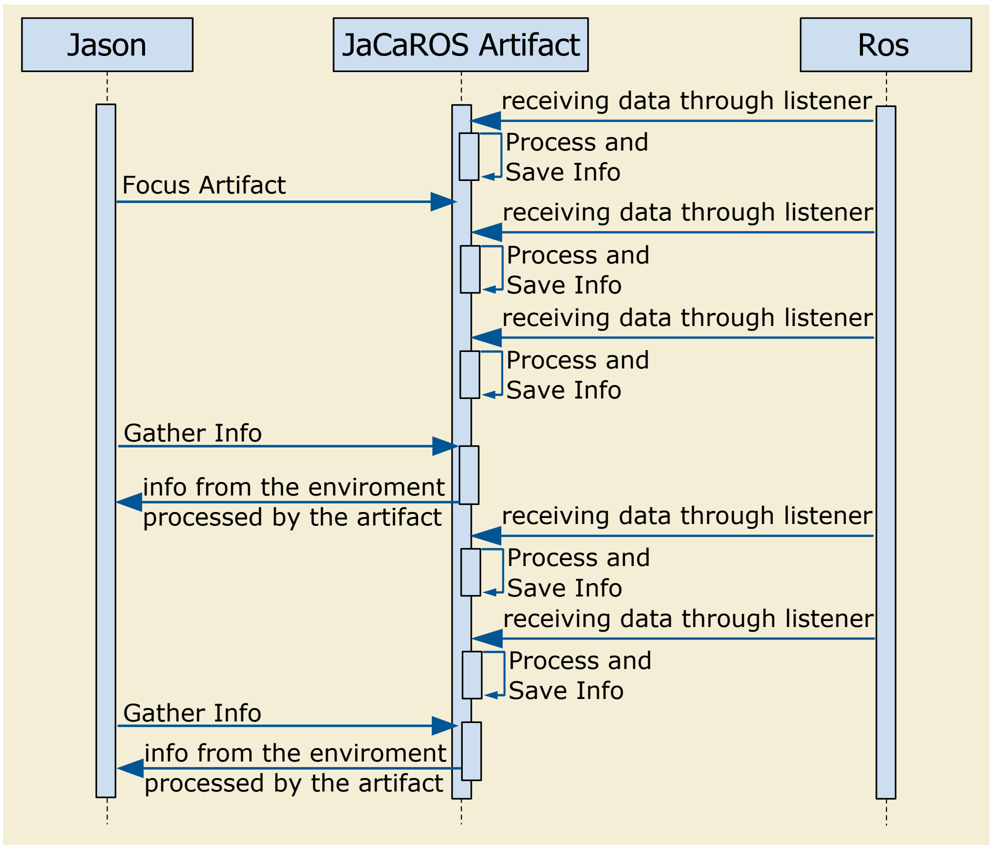
Jason
JaCaROS Artifact
Ros
receiving data through listener
Process and Save Info
Focus Artifact
Gather Info
info from the enviroment processed by the artifact

Figure 7.4 – Sequence diagram of a perception from the environment, on demand mode.

7.2.4 Publisher Artifact

The following section presents how a JaCaROS artifact which uses a *publisher* is developed: in order to publish data into ROS topic, the Jason agent needs to use the artifacts operations. In our example (Listing 16), through the use of *moveForward(double distance)* operation (Line 1), the agent is able to publish linear data (Line 18) into the ROS topic and as a consequence, move the robot into a given distance. In Line 3, the artifact defines that it want to apply a movement into the X axis, using a given speed. All the other axes are defined to have no speed; Line 8 saves the current position of the robot and uses it in the future to calculate the traveled distance; While the distance is not reached and while the abort flag is false (Line 13), the artifact keep publishing movement into the ROS topic (Line 18).

Figure 7.5 shows the processes sequence that must be done to publish in a given topic and consequently, interact with ROS actuators: first, the Jason agent calls an artifact operation (such as rotate or move forward); Second, the JaCaROS artifact executes the desired operation, publishing a message in the correct ROS topic. Artifacts may keep publishing messages on the topic until a given condition is satisfied and this stop condition may be part of the information given by the agent

```
@OPERATION void moveForward(double distance){
  twist = publisherCmdVel.newMessage(); //initializing variable
  twist.getLinear().setX(speed); twist.getLinear().setY(0); twist.getLinear().setZ(0);
  twist.getAngular().setX(0); twist.getAngular().setY(0); twist.getAngular().setZ(0);
  this.abort = false; //initialize abort flag
  double initialPose; //initializing variable
  if (currentOdometry!=null)
   initialPose = currentOdometry.getX();
  else
   initialPose = 0;

  stepCounter = 0; //initialize stepcounter
  while (!this.abort && stepCounter <= distance)
  {
   if (currentOdometry!=null)
    stepCounter = currentOdometry.getX() - initialPose;

   publisherCmdVel.publish(twist);
  }
 }
 @OPERATION void abortMoving() {
  this.abort = true;
 }
```

Listing 16: ArtCmdVel artifact pseudo code: Publishing movement into a ROS topic.

when it calls the operation. Meanwhile, an artifact can receive information from a given topic and it may use this information as the stop condition.

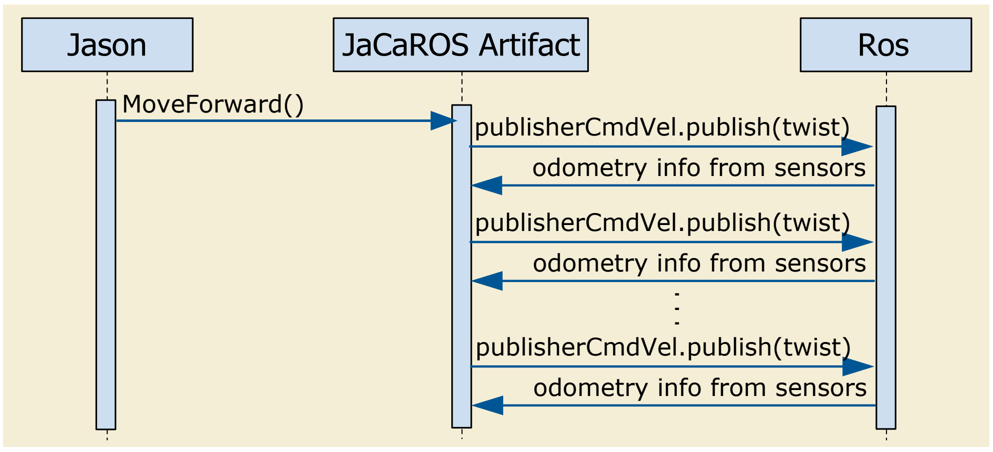
Jason
JaCaROS Artifact
Ros
MoveForward()
publisherCmdVel.publish(twist)
odometry info from sensors
publisherCmdVel.publish(twist)
odometry info from sensors
publisherCmdVel.publish(twist)
odometry info from sensors

Figure 7.5 – Sequence diagram of an operation in an actuator (move forward example).

7.2.5 Client Artifact

This section presents how a JaCaROS artifact which interacts with a *service* is developed. We used ArtTutorial artifact as an example because basically, all the artifacts which interact with a ROS service work in a similar way. In our example (Listing 17), through the use of *sum* operation (Line 1), the agent is able to interact with ROS service. In Line 2, the artifact creates a new variable (*request*) responsible to request the service from ROS; In Line 3 and Line 4, the artifact sets the input values needed by the service; In Line 5, the artifact request the service, creating a new Listener responsible to receive the ROS service response; if the request was successful (Line 7), the artifact saves the response in a local variable (Line 8) and after that, runs an internal action called *receivingSum* (Line 9). This internal action is responsible to update the observable property and as a consequence, the information from the service is propagated to the agent.

```
 1  @OPERATION void sum(int valueA, int valueB){
 2    final AddTwoIntsRequest request = serviceClient.newMessage();
 3    request.setA(valueA);
 4    request.setB(valueB);
 5    serviceClient.call(request, new ServiceResponseListener<AddTwoIntsResponse>() {
 6      @Override
 7      public void onSuccess(AddTwoIntsResponse response) {
 8        currentSum = response.getSum();
 9        execInternalOp("receivingSum");
10        }
11      @Override
12      public void onFailure(RemoteException e) {
13        currentSum = 0;
14      }
15    });
16  }
```

Listing 17: ArtTutorial artifact pseudo code: interaction with a ROS service.

The use of a listener causes inversion of control - as we explained before in section 7.2.2 - thus, we implement a IBlockingCmd class to certify that the update of the observable property will be performed within the correct thread, as shown in Listing 18.

```
...
private ReadCmdSum cmdSum;
...
@INTERNAL_OPERATION
void receivingSum() {
  await(cmdSum);
  signal(propertyNameSum);
  }
class ReadCmdSum implements IBlockingCmd {
  public void exec() {
    try {
      ObsProperty prop = getObsProperty(propertyNameSum);
      prop.updateValues(currentSum);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Listing 18: ArtTutorial artifact pseudo code: internal operation.

8. EXPERIMENTS

In order to understand how JaCaROS simplify the development of complex behaviors on robots, we compare the code developed using Python - a common language used to develop robot algorithms - with the code developed using JaCaROS for a set of different scenarios. After that, we analyzed the resolution strategy and the source code of both approaches. The Python code used in the experiments were developed by Patrick Goebel and extracted from the books ROS by Example [16][17], while Jason code and Java code were developed by ourselves.

Since there are usually more than one way to solve a given problem, the pseudo-code blocks that we are presented in this chapter are only meant as a guide, and may not represent the best way (or even the only way) to perform a given goal at illustrated scenarios.

8.1 Experimental Design

In this Section, we provide a high-level description of five scenarios in which a Turtlebot[1] robot interacts with a simulated environment and performs tasks. The scenarios were inspired on the books ROS by Example [16][17], and we describe them below:

The time-based out-and-back Scenario: in this scenario, a robot should move forward during a given time, turn around 180 degrees, then come back to the starting point, rotating the robot 180 degrees one more time to match with the original orientation, as presented on Figure 8.1.

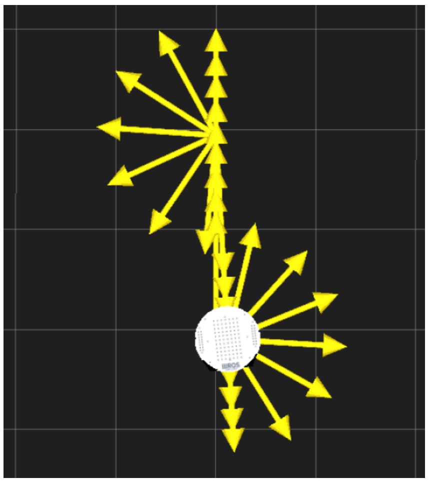

Figure 8.1 – The Time-Based Out-and-Back Scenario [16].

Navigating a square scenario: in this scenario, a robot should move the perimeter of a square by navigating from corner to corner, in sequence. Each side of the square has one meter. The robot should come back to the starting point at the same position and orientation that it has started, as presented on Figure 8.2.

[1]Turtlebot tobot is presented in Section APPENDIX D

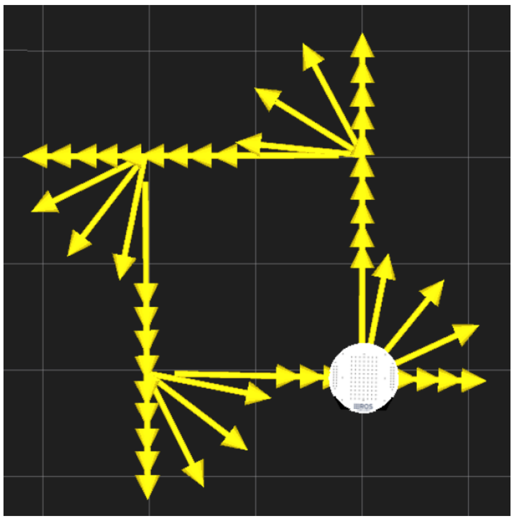

Figure 8.2 – Navigating a Square Scenario [16].

The fake battery simulator: in this scenario, we simulate the behavior of a mobile robot battery. The scenario starts with the initial battery level set to 100 (simulating 100% of battery), and its counting down to 0 over a time period. Diagnostics status level are defined, such as Full Battery, Medium Battery (when the counter reaches 50) and Low Battery (when the counter reaches 20). The agent should know the current diagnostic status level.

The patrol robot scenario: in this scenario, a robot should patrol the perimeter of a square by navigating from corner to corner, in sequence. The robot should monitor the battery level and if it reaches a given value, the robot should stop patrolling and recharge. After recharging, the robot should continue to patrol the perimeter of the square, from where it left off. In Figure 8.3, the waypoints are illustrated as colored squares and the recharge point (the docking station) as a yellow disc.

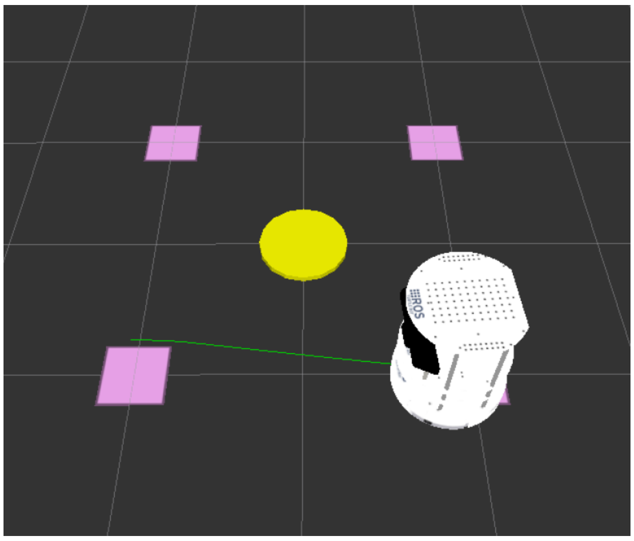

Figure 8.3 – The Patrol Robot Scenario [17].

The house cleaning robot Scenario: in this scenario, we suppose that each corner of a square represents a room, and the robot should perform one or more cleaning tasks in each room.

When the robot finished the cleaning task, it should move to the next room. The square corners represent a living room, a kitchen, a bathroom and a hallway, and the tasks for each room are the following: in the living room, the robot should vacuum the carpet; in the kitchen, it should mop the floor; and in the bathroom it should perform two tasks: scrub the tub and mop the floor.

8.2 Experimental Development

8.2.1 The Time-Based Out-and-Back Scenario

The code used to accomplish the current scenario using JaCaROS architecture is presented in Listing 19 in addition to *ArtCmdVel* artifact presented in Listing APPENDIX B.6. The Jason code is the following: between Line 3 and Line 14, we have the plans to create and use the *ArtCmdVel* JaCaROS artifact; between Line 17 and Line 19, we use these plans; and finally, between Line 20 and Line 22, we move the robot in order to accomplish the current scenario goal.

```
 1 !start.
 2 
 3 +!create_cmd_vel
 4   <- makeArtifact("Cmd_vel","jason.architecture.ArtCmdVel",[],CmdVelId).
 5 
 6 -!create_cmd_vel
 7   <- .print("cmd_vel artifact creation failed.").
 8 
 9 +?toolCmd_vel(CmdVelId)
10   <- lookupArtifact("Cmd_vel", CmdVelId).
11 
12 -?toolCmd_vel(CmdVelId)
13   <- .wait(10);
14      ?toolCmd_vel(CmdVelId).
15 
16 +!start
17   <- !create_cmd_vel;
18      ?toolCmd_vel(CmdVelId);
19      focus(CmdVelId);
20      for ( .range(I,1,2) ) { //go out and return
21         move(25); //period spend publishing
22         rotate(180);
23      }.
```

Listing 19: Jason code using JaCaROS for the time-based out-and-back scenario.

The artifact code (Listing APPENDIX B.6) is the following: the code configures a publisher in Line 21; the movement speed is set in Line 7; the amount of time for publishing the forward movement comes from a parameter of the artifact operation, as shown in Line 25 (*linear_duration*); the angle degree comes from a parameter of the artifact operation, as shown in Line 35 (*degree*); the forward movement is done between Line 29 and Line 32; and the rotate movement is done between Line 45 and Line 48;

The python code used to accomplish the current scenario is presented in Listing APPENDIX C.1. The code configures a publisher in Line 10; the amount of time for publishing the forward movement is set on Line 15; the amount of time for publishing the rotate movement is set on Line 18; the forward movement is done between Line 26 and Line 28; the rotate movement is done between Line 41 and 43; The script finally has methods responsible for shutdowning the robot, between Line 53 and 57;

8.2.2 Navigating a Square Scenario

The code used to accomplish the current scenario using JaCaROS architecture is presented in Listing 20 in addition to *ArtCmdVelMuxAndOdometry* artifact presented in Listing APPENDIX B.7. This Jason code is almost the same comparing with the last scenario. The differences are: first, we are using a artifact in which uses odometry to define when to stop publishing the forward movement, instead a period of time; and second, between the Line 20 and Line 22, we change the rotate angle and the number of times that the movement repeats, in order to accomplish the current scenario goal (navigate a square).

```
1  !start.
2
3  +!create_cmd_vel_mux_odom
4    <- makeArtifact("Cmd_vel_mux_odom","jason.architecture.ArtCmdVelMuxAndOdometry",[],
        CmdVelMuxId).
5
6  -!create_cmd_vel_mux_odom
7    <- .print("cmd_vel_mux_odom artifact creation failed.").
8
9  +?toolCmd_vel_mux_odom(CmdVelMuxId)
10   <- lookupArtifact("Cmd_vel_mux_odom",CmdVelMuxId).
11
12 -?toolCmd_vel_mux_odom(CmdVelMuxId)
13   <- .wait(10);
14      ?toolCmd_vel_mux_odom(CmdVelMuxId).
15
16 +!start
17   <- !create_cmd_vel_mux_odom;
18      ?toolCmd_vel_mux_odom(CmdVelMuxId);
19      focus(CmdVelMuxId);
20      for ( .range(I,1,4) ) { //4 sides of the square
21        moveForward(1); //distance in meters
22        rotate(90);
23   }.
```

Listing 20: Jason code using JaCaROS for the navigating a square scenario.

The artifact code used in this scenario (Listing APPENDIX B.7) performs the following: it configures a publisher in Line 32 (in order to give movement to the robot), and it configures a subscriber in Line 36 (in order to receive odometry information); speed is set in Line 9 and used both for linear and angular motion; the distance of one meter comes from a parameter of the artifact operation (*distance*), as shown in Line 53; the angle degree comes from a parameter of the artifact operation (*degree*), as shown in Line 76; the forward movement is done between Line 64 and Line 70; and finally, the rotate movement is done between Line 87 and Line 90.

The python code used to accomplish the current scenario is presented in Listing APPENDIX C.2. Between the Line 16 and Line 19, linear speed, angular speed, distance (one meter) and angle degree (90 degrees) are set; a publisher is set in Line 22; a odometry listener is configured in Line 25; The script cycles through the four sides of the square by using a *for* in Line 43 and keep publishing on ROS topic until one meter is reached between Line 54 and Line 63; the rotate movement is done between Line 75 and Line 82; and finally, the script has methods responsible for shutdowning the robot, between Line 100 and Line 104;

8.2.3 The Fake Battery Simulator

The code used to satisfy the current scenario using JaCaROS architecture is presented in Listing 21 in addition to the artifact presented in Listing APPENDIX B.8. The Jason code is the following: between Line 3 and Line 14, we have the plans to create and use the *ArtSimBattery* artifact; between Line 21 and Line 23, we use these plans; and finally, between Line 16 and Line 18, we print the diagnostic message when the agent get the belief (*diagnostics*) from the artifact observed property.

```
1  !start.
2  
3  +!create_battery <-
4    makeArtifact("Battery","jason.architecture.ArtSimBattery",[],BatteryId).
5  
6  -!create_battery <-
7    .print("Battery artifact creation failed.").
8  
9  +?toolBattery(BatteryId) <-
10   lookupArtifact("Battery",BatteryId).
11 
12 -?toolBattery(BatteryId) <-
13   .wait(10);
14   ?toolBattery(BatteryId).
15 
16 +diagnostics(Diag) <-
17   .print(Diag);
18   -diagnostics(Diag).
19 
20 +!start <-
21   !create_battery;
22   ?toolBattery(BatteryId);
23   focus(BatteryId).
```
Listing 21: Jason code for fake battery scenario.

The artifact code used in this scenario (Listing APPENDIX B.8) do the following: the diagnostic levels are defined between Line 8 and Line 10; step size used to decrease the battery level is set in Line 11; a property responsible for handling battery level is created in Line 14; a property responsible for handling diagnostics is created in Line 16; between Line 53 and Line 63, the artifact sets the diagnostics status level based on the current battery level and the property is updated in Line 66; and a *timer* in Line 20 is responsible for decreasing battery level, in Line 23. If the agent want to recharge battery, this can be done by using the *operationRecharge()* operation (Line 70).

The python code used to accomplish the current scenario is presented in Listing APPENDIX C.3: the diagnostic levels are defined between Line 15 and Line 17; step size used to decrease the battery level is set in Line 20; a battery level publisher is created in Line 22; a diagnostic publisher is created in Line 24; between Line 35 and Line 43, the python code set the diagnostics status level based on the current battery level; and method *SetBatteryLevelHandler* in Line 69 is responsible for changing the battery level value.

8.2.4 The Patrol Robot Scenario

The code used to satisfy the current scenario using JaCaROS architecture is presented in Listing 22 and Listing 23. We split it out in two listings in order to make the code clear. Let's start explaining Listing 22: between Line 10 and Line 16, the agent creates and focuses the artifacts ArtCmdVelMuxAndOdometry (in order to move the agent) and ArtSimBattery (in order to check simulated battery). Plans for create and use these artifacts were omitted since previous pseudo-code blocks already illustrate them. In fact, the plans used to create and to use a given artifact are reusable, it can be saved in a separate file and included in the agent's code when needed; In Line 17, the agent adopts a new goal (to patrol); As shown between Line 4 and Line 7, the *patrol* plan is composed of a 2 meters forward movement plus a 90 degree rotation. The agent will keep patrolling in a loop since *patrol* is a recursive plan, as shown in Line 7.

```
 1  /* Plans for create and use ArtSimBattery and ArtCmdVelMuxAndOdometry were ommited */
 2  !start.
 3
 4  +!patrol <-
 5         moveForward(2);
 6         rotate(90);
 7         !!patrol. // recursive call.
 8
 9  +!start <-
10    !create_cmd_vel_mux_odom;
11    ?toolCmd_vel_mux_odom(CmdVelMuxId);
12    focus(CmdVelMuxId);
13
14    !create_battery;
15    ?toolBattery(BatId);
16    focus(BatId);
17    !patrol.
```

Listing 22: Jason pseudo-code using JaCaROS for the patrol robot scenario.

As shown in Listing 23, the *recharge* plan (between Line 15 and Line 20) is composed of moving to the center of square and after that, executing an artifact operation: *operationRecharge*, Line 19. This operation is responsible for recharging the battery; when finishing recharging, the agent returns (Line 20) to the previous square corner, as shown between Lines 22 and 27. The agent suspends (Line 12) the *patrol* plan when it receives an update of "Low Battery" from the observed property, as shown between Line 9 and Line 13; and it resumes the *patrol* plan (Line 28) after return to last square corner visited before go recharging.

```
 1  +diagnostics(Diag) : Diag = "Full Battery" <-
 2    .print("Current Diagnostic: ",Diag);
 3    -diagnostics(Diag).
 4
 5  +diagnostics(Diag) : Diag = "Medium Battery" <-
 6    .print("Current Diagnostic: ",Diag);
 7    -diagnostics(Diag).
 8
 9  +diagnostics(Diag) : Diag = "Low Battery" <-
10    .print("Current Diagnostic: ",Diag);
11    -diagnostics(Diag);
12    .suspend(patrol); //Suspending Patrol plan
13    !recharge.
14
15  +!recharge <-
16    moveForward(1);
17    rotate(90);
18    moveForward(1); //Now agent arrived in Recharge Base
19    operationRecharge;
20    !return. //Return to previous square corner
21
22  +!return <-
23    rotate(180); //Turned around
24    moveForward(1);
25    rotate(-90);
26    moveForward(1); //original position
27    rotate(180); //original orientation
28    .resume(patrol).
```

Listing 23: Jason pseudo-code using JaCaROS for the patrol robot scenario.

To satisfy current scenario goals, two artifacts were used: *ArtSimBattery* to simulate the battery and *ArtCmdVelMuxAndOdometry* (to move the robot). Both artifacts were previously used and explained in the last scenarios.

The patrol robot scenario requires more reasoning than the three previous scenarios. The robot should perform two distinct tasks at same time: patrol the corners of a square and keep track of battery level. If the battery level falls below a certain level, the robot should stop its patrol and navigate to the docking station. All of this can be done using ROS actions, but despite this works for this particular example, it becomes less efficient as we add more tasks to the robot's behavior [17]. Thus, in order to create a code to fulfill the current scenario, the python developer may need to simulate the interpretation cycle and develop robot behaviors using *hierarchical state machines* or *behavior trees*. Listing APPENDIX C.5 and Listing APPENDIX C.6 show the pseudo code of the libraries used in this scenario to create and use behavior trees for Python and for ROS, but explain them is beyond the scope of this work.

The python code used to accomplish the current scenario is presented in Listing APPENDIX C.7: Between Line 18 and Line 24, a navigation task is created for each square corner; The local of the recharge place (docking station) is set between Line 27 and Line 30; A motion action to the docking station is added in Line 33; A behavior tree is created with the tasks STAY_HEALTHY and LOOP_PATROL between Line 40 and Line 41; in Line 49, patrol task is added on a loop; battery check and recharge tasks are added into the STAY_HEALTHY task between Line 52 and Line 61; Finally, the tree runs between Line 68 and Line 70.

8.2.5 The House Cleaning Robot Scenario

The code used to fulfill the current scenario using JaCaROS architecture is presented in Listing 24 and Listing 25, in addition to the artifact presented in Listing APPENDIX B.9. Jason code is based on the source code of the previous one, thus, we present in Listing 24 and Listing 25 just the differences between them. Let's start explaining Listing 24.

```
1  !start.
2
3  +!create_cleaner <-
4    makeArtifact("Cleaner","jason.architecture.ArtCleaning",[],Cleaner).
5
6  -!create_cleaner <-
7    .print("Cleaner artifact creation failed.").
8
9  +?toolCleaner(CleanerId) <-
10   lookupArtifact("Cleaner",CleanerId).
11
12 -?toolCleaner(CleanerId) <-
13   .wait(10);
14   ?toolCleaner(CleanerId).
15
16 +!patrol <-
17   moveForward(2);
18   rotate(90);
19   !cleanRoom; //clean room
20   !!patrol.
21
22 +!start
23   ...
24   !create_cleaner;
25   ?toolCleaner(CleanerId);
26   focus(CleanerId);
27   ...
28   !patrol.
```

Listing 24: Jason pseudo-code using JaCaROS for the house cleaning robot simulator.

Plans to create and use the *ArtCleaning* artifact is shown between Line 3 and Line 14, and these plans are used between Line 24 and Line 26. The *patrol* plan (Line 16) acquired the *cleanRoom* goal in Line 19 and the plan used to reach this goal is responsible for cleaning the current room.

As we can see in Listing 25, between Line 13 and Line 19, *cleanRoom* plan is responsible for verifying the current room (*updateWhereAmI* operation, Line 15) and ask the artifact for a task (vacuum, mop or scrub), through the use of the *clean* operation, Line 17. The clean operation changes the value of observable properties and as a consequence, may activate a set of triggers (between Lines 1 and 11). The tasks (vacuum, mop or scrub) are represented by messages, as we can see in Line 2, Line 6 and Line 10, and the cleaning activity is simulated as a swing (Line 18). In the current scenario, we need to suspend *cleanRoom* plan (Line 24) before recharge (Line 25) and resume *cleanRoom* plan (Line 30) after return to last place visited before go recharging (Line 29).

```
 1  +vacuum(Object, Room) <-
 2    .print("At: ", Room); .println("Vacuum ", Object);
 3    -vacuum(Object, Room).
 4
 5  +mop(Object, Room) <-
 6    .print("At ", Room); .println("Mop ", Object);
 7    -mop(Object, Room).
 8
 9  +scrub(Object, Room) <-
10    .print("At ", Room); .println("Scrub ", Object);
11    -scrub(Object, Room).
12
13  +!cleanRoom <-
14    -atRoom(Any);
15    updateWhereAmI; //update current local
16    ?atRoom(Y);
17    clean(Y); //clean room
18    rotate(-5); rotate(10); rotate(-5); //swing a little to simulated clean activity,
19    .wait(1000). //simulate clean time spend
20
21  +diagnostics(Diag) : Diag = "Low Battery" <-
22    ..
23    .suspend(patrol); //Suspending Patrol plan
24    .suspend(cleanRoom); //Suspending Clean plan
25    !recharge.
26
27  +!return <-
28   ...
29    rotate(180); //original orientation
30    .resume(cleanRoom);
31    .resume(patrol).
```

Listing 25: Jason pseudo-code using JaCaROS for the house cleaning robot simulator.

To fulfill this scenario, tree artifacts were used: *ArtSimBattery* (to simulate the battery), *ArtCmdVelMuxAndOdometry* (to move the robot) and *ArtCleaning* (to clean the rooms). *ArtSim-Battery* and *ArtCmdVelMuxAndOdometry* were previously used and explained in the last scenarios, thus we will focus our attention on ArtCleaning artifact (Listing APPENDIX B.9): Properties for vacuum, mop and scrub are created between Lines 3 and 5; the room names are defined between Lines 7 and 10; the room that the agent is current patrolling is a parameter of *clean* operation (Line 29), and this operation updates the observed properties according to the room (such as vacuum the living room carpet or mop the kitchen floor).

The current scenario is another situation where more reasoning from the agent is needed. Thus, in order to satisfy this scenario, the python code developer may need to create *behavior trees*. The libraries from the last scenario (Listing APPENDIX C.5 and Listing APPENDIX C.6) are used in python pseudo-code of the current scenario. It is divided in two pseudo-code blocks: Listing APPENDIX C.8 and Listing APPENDIX C.9. Listing APPENDIX C.9 contains the simulated cleaning tasks: the vacuum task in Line 6, the mop task in Line 38 and the scrub task in line 70. Each of them simulate the clean activity by short strokes in the robot (Line 25, Line 58 and Line 90). Listing APPENDIX C.8 have code in common with the last scenario source code, thus we focus on the differences. Between Line 26 and Line 31, the room names are defined and tasks are mapped for each room. For each room, a set of activities are performed between Line 131 and Line 173, such as: check current room (Line 136); initialize the cleaning task (Line 140); initialize the navigation task (Line 142); add move base task and navigate to room task to the behavior three (between Line 147 and Line 150); update task list (between Line 159 and Line 166); and finally the cleaning activities (between Line 168 and Line 173).

8.3 Discussion

Below we have three code metrics, obtained from static analysis of the source code used to program the scenarios. Table 8.1 shows the number of identifiers used by each approach. For the Jason code, we considered plans, triggers and operations as identifiers and for Java and Python code, we considered classes, methods and variables. Table 8.2 shows the number of lines of code needed to program the behaviors using each approach. Although Lines of Code (LOC) are not an unanimous metric to measure code quality, we believe it is an acceptable estimation of code complexity. In particular, since we want to measure the complexity resulting from the different abstraction levels, we use this metric in our experiments. According to Kan [23, chapter 11, pg 312], LOC count represents the program size and its complexity. Finally, Table 8.3 shows the measure of source code size by compressing it with *gzip*.

Environment	Complexity	JaCaROS			Python
		Jason	Artifacts	Total	
Time-Based Out-and-Back	Low	8	17	25	19
Navigating a Square	Low	8	28	36	27
The Fake Battery Simulator	Low	7	21	28	23
The Patrol Robot	Medium	31	49	80	143
The House Cleaning Robot	High	52	65	117	173

Table 8.1 – Code metrics: static analysis of both approaches. Number of Identifiers.

Environment	Complexity	JaCaROS			Python
		Jason	Artifacts	Total	
Time-Based Out-and-Back	Low	23	50	73	63
Navigating a Square	Low	23	107	130	110
The Fake Battery Simulator	Low	23	81	104	76
The Patrol Robot	Medium	73	188	261	536
The House Cleaning Robot	High	127	255	382	759

Table 8.2 – Code metrics: static analysis of both approaches. Lines of code.

Environment	Complexity	JaCaROS			Python
		Jason	Artifacts	Total	
Time-Based Out-and-Back	Low	289	632	921	890
Navigating a Square	Low	311	1.239	1550	1.392
The Fake Battery Simulator	Low	230	665	895	1.196
The Patrol Robot	Medium	527	1.721	2248	4.215
The House Cleaning Robot	High	791	2.110	2901	6.443

Table 8.3 – Code metrics: static analysis of both approaches. Source code size (bytes) compressed by gzip.

The robot within our first three scenarios has a simple behavior. Basically, at *time-based out-and-back* scenario and at *navigating a square* scenario, the robot just walks around in the environment; and at the fake battery scenario, the robot just need to be aware of the battery level. On the other hand, the robot within last two scenarios has a more complex behavior, since it should perform distinct tasks at same time, such as: walk around, check battery and cleaning. A complex behavior requires more reasoning from the robot, and more code development.

For all five scenarios, the Jason code has less identifiers and less LOCs than the Python code. But when we take JaCaROS artifacts into consideration, we noticed a difference in the results obtained in simple scenarios compared to complex scenarios: in simple scenarios, Jason code plus artifact code have slightly more identifiers and lines of code than Python. But, when the development of complex behaviors are needed, Jason code plus artifact code have the expected results: less identifiers and less lines of code compared to Python, as we can see in Table 8.1 and Table 8.2. The same behavior is detected when we analyze the size of source code compressed by gzip, as shown in Table 8.3.

Figure 8.4, Figure 8.5 and Figure 8.6 show relevant part of JaCaROS code and Python code for the first three scenarios. As we can verify, the code for each scenario is very similar to one another. Python was able to accomplish the scenario goals with a similar abstraction level as JaCaROS and as a result, our architecture has not made a significant difference to simplify the development of the code.

The robot within our last two scenarios has a complex behavior. The scenario complexity increasing results in the addition of new tasks in the Python code, increases the number of events to handle, and may result in a Python source code "polluted" by large blocks of if-then-else statements (or any conditional statements, such as switch case). In order to develop the source code for *the patrol robot* scenario and *the house cleaning robot* scenario in Python, an alternative approach called behavior trees was used. Behavior tree is a model for plan execution which enables the code developer to decompose tasks into a set of in a subtasks, in a modular fashion [17]. The tree is always executed from top to bottom and left to right; higher nodes in the tree are more abstract than lower levels and just terminal nodes in each branch of the tree result in actual behavior.

For the last two scenarios, we are not able to easily compare and contrast bits of code for JaCarROS and Python, in order to highlight the differences, since the creation of behavior trees

Jason

```
for ( .range(I,1,2) ) {
  move(25);
  rotate(180);
```

Artifact

```
for (int a=0; a<=linear_duration; a++){
  publisherCmdVel.publish(twist);
  sleepNoLog(100);
}
...
for (short i=0; i<degree/15; i++) {
  publisherCmdVel.publish(twist);
  sleepNoLog(100);
  }
```

Python

```
for i in range(2):
    move_cmd = Twist() # Initialize the movement command
    move_cmd.linear.x = linear_speed # Set the forward speed
    ticks = int(linear_duration * rate) # Move forward for a
    for t in range(ticks):               # time to go
        self.cmd_vel.publish(move_cmd)   # the desired distance
        r.sleep()

    move_cmd.angular.z = angular_speed
    ticks = int(goal angle * rate) # Rotate for a time to go
    for t in range(ticks):                   # 180 degrees
        self.cmd_vel.publish(move_cmd)
        r.sleep()
```

Figure 8.4 – Pseudo-code blocks for Time-Based Out-and-Back Scenario.

Jason

```
for ( .range(I,1,4) ) {
  moveForward(1);
  rotate(90);
```

Artifact

```
stepCounter = 0; //initialize stepcounter
while (!this.abort && stepCounter <= distance) {
  if (currentOdometry!=null)
    stepCounter = currentOdometry.getX() - initialPose;
  sleepNoLog(10);
  publisherCmdVelMux.publish(twist);
}
...
for (short i=0; i<degree/15; i++)
    publisherCmdVelMux.publish(twist);
    sleepNoLog(10);
  }
}
```

Python

```
# Cycle through the four sides of the square
for i in range(4):
...
    distance = 0 # Keep track of the distance traveled
    # Enter the loop to move along a side
    while distance < goal_distance and not rospy.is_shutdown():
        # Publish the Twist message and sleep 1 cycle
        self.cmd_vel.publish(move_cmd)
        r.sleep()
        (position, rotation) = self.get_odom() # Get the current position
        # Compute the Euclidean distance from the start
        distance = sqrt(pow((position.x - x_start), 2) +
                        pow((position.y - y_start), 2))
    ...
    # Begin the rotation
    while abs(turn_angle + angular_tolerance) < abs(goal_angle) and not rospy.
        is_shutdown():
        self.cmd_vel.publish(move_cmd) # Publish the Twist message and sleep 1 cycle
        r.sleep()
    ...
```

Figure 8.5 – Pseudo-code blocks for Navigating a Square Scenario.

Jason

```
+diagnostics(Diag) <-
  .print(Diag);
  -diagnostics(Diag).
```

Artifact

```
switch (currentBattery) {
  case FULL:
    currentDiagnostics = "Full Battery";
    break;
  case MEDIUM:
    currentDiagnostics = "Medium Battery";
    break;
  case LOW:
    currentDiagnostics = "Low Battery";
    break;
}
...
```

Python

```
...
# Set the diagnostics status level based on the current battery level
if self.current_battery_level < self.error_battery_level:
  status.message = "Low Battery"
  status.level = DiagnosticStatus.ERROR
elif self.current_battery_level < self.warn_battery_level:
  status.message = "Medium Battery"
  status.level = DiagnosticStatus.WARN
else:
  status.message = "Battery OK"
  status.level = DiagnosticStatus.OK
...
```

Figure 8.6 – Pseudo-code blocks for The Fake Battery Simulator.

makes Python code very different from our architecture code. Therefore, for these two scenarios, we are just analyzing the results of code metrics. The use of behavior trees can make easier the development of rational behaviors for classical programming languages, but does not prevent the increase in code complexity, proportional to the rationality necessary to accomplish a given task or behavior at the presented scenarios. The more the complexity of the scene grows, the greater is the number of identifiers and greater is the number of lines of code in Python. On the other hand, despite the increase of robot rational behavior at the last two scenarios, the complexity of Jason code and JaCaROS artifacts code did not increase in the same proportion that Python code increased, as we can see in Figure 8.7, Figure 8.8 and Figure 8.9.

The disconnection between the agents layer - where the agent code is developed - and the interface layer - where the JaCaROS artifacts are developed provided by our architecture contributes for the development of robot complex behaviors in a simpler way, since this disconnection facilitates modularization and code reuse: Jason plans to create and use artifacts and the artifacts themselves were reused in the presented scenarios.

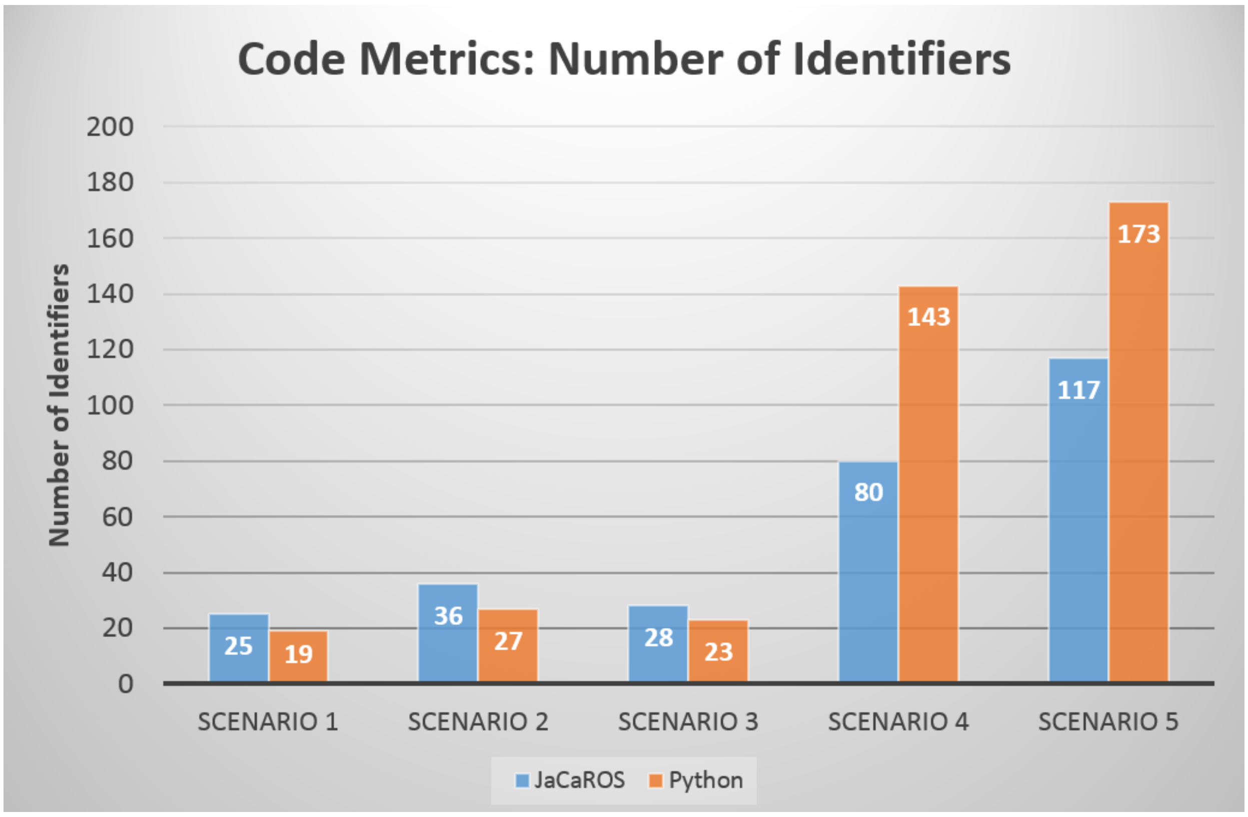

Figure 8.7 – Code Analysis: Number of Identifiers by scenario.

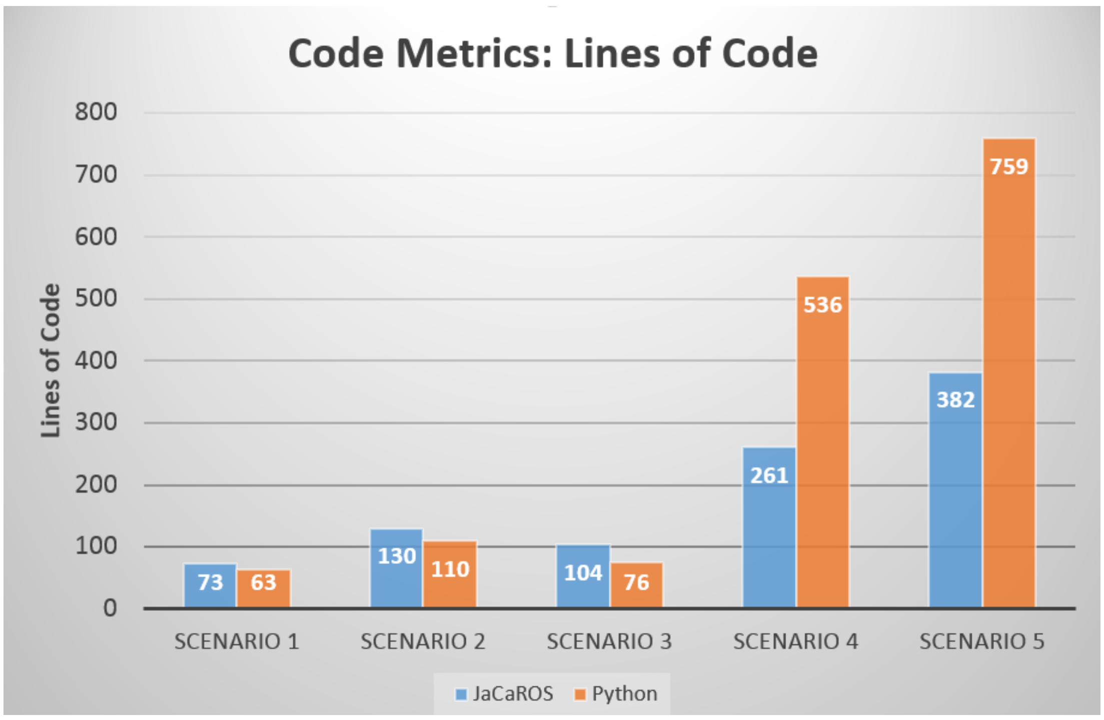

Figure 8.8 – Code Analysis: Lines of Code by scenario.

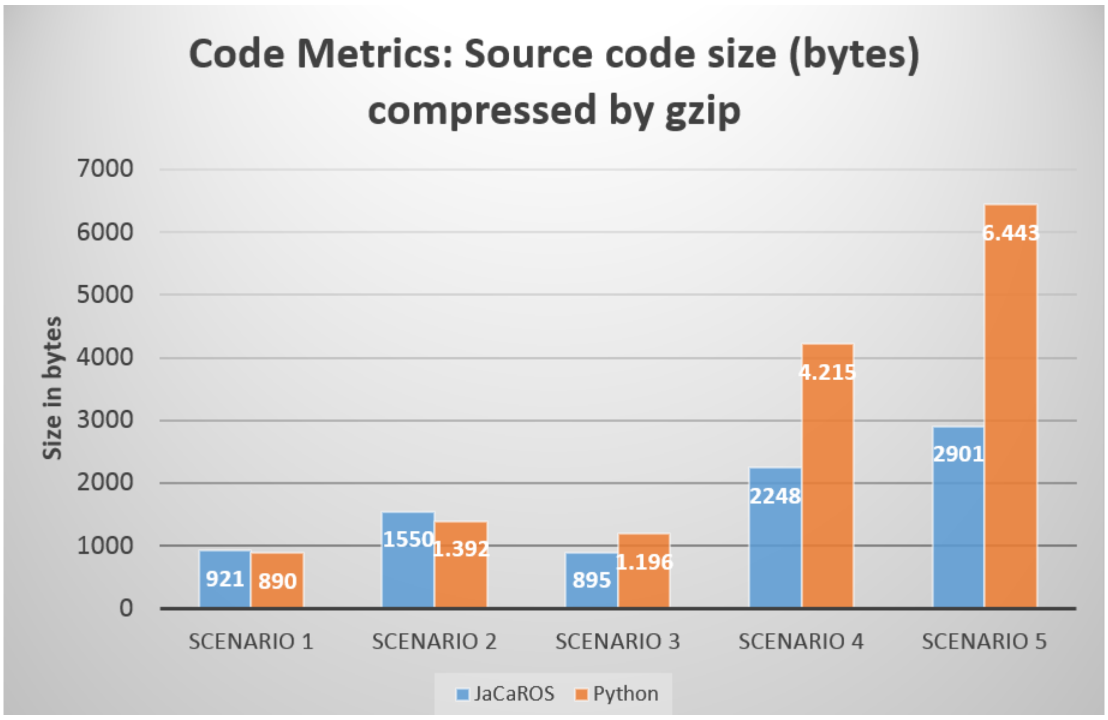

Figure 8.9 – Code Analysis: Code Size in bytes, compressed by gzip.

9. RELATED WORK

Ziafati [57] presents four requirements for BDI-based agent programming languages to facilitate the implementation of autonomous robot control systems. The requirements are the following: built-in support for integration with existing robotic frameworks; real-time reactivity to events; management of heterogeneous sensory data and reasoning on complex events; and finally representation of complex plans and coordination of the parallel execution of plans. JaCaROS satisfies the first requirement using its interface layer, Jason and ROS; the use of a *verbose mode* artifact helps the second requirement to be reached; heterogeneous sensory data can be handled by the JaCaROS artifacts and translate to meaningful information to the agent. Furthermore, the use of Jason proved to be a better approach handling complex behaviors, helping to satisfy the third requirement; finally, the Jason language supports the development of complex plans helping to reach the last requirement. As a proof of concept, Ziafati developed an environment interface for 2APL to simplify its integration with ROS, enabling the communication of 2APL with ROS components using ROS communication mechanisms.

Verbeek [49] research extends the 3APL language for a high level robot control by creating a communication interface between the 3APL and ROS. According to Verbeek, the basic actions in 3APL are only capable of mental belief updates, thus the language needs to be extended with external basic actions that will be executed by an external program (in this case, ROS). Experiments were conducted in a simulated and in a physical environment, focused on the accuracy of the movements and localization. Currently, we have artifacts able to make a mobile robot move within physical environments and at simulated environments, and our *SLAM* artifact is able to make a mobile robot gather, save and retrieve localization information. This artifact allows an agent to reason about its own position both in relative terms (how far it moved) and in absolute terms (where exactly in the map the agent, other agents or objects are). This artifact is responsible for two general functions: store a grid representation of the map, including the current position of the agent, free positions, obstacle positions and special positions (default keywords), such as "home" and "goal"; and provide the correct direction that an agent must follow to reach a given goal. In order to store a grid representation of the map, we use a Cartesian coordinate system, adopting a format to represent the environment positions: *pos2d(Object,Xpos,Ypos)*, where *pos2d* represents a 2D position tuple; *Xpos* represents the position at X axis; *Ypos* represents the position at Y axis; and *Object* represents the object name located at Xpos and Ypos position.

Santi et al. [42] discuss the application of an agent-oriented programming platform (in this case, JaCa) for the development of smart mobile applications. In order to apply JaCa to mobile computing, the authors developed a porting of the platform on top of Google Android, where a mobile application can be realized as a workspace in which Jason agents are used to encapsulate the logic and the control of mobile application tasks, and artifacts are used as tools for agents to exploit available Android components and services. Namely, they extended JaCa with a predefined set of artifacts specifically designed for exploiting Android functionalities. According to Santi et al.

[42], the motivation to use JaCa on Android is address issues such as concurrency, asynchronous interactions with different kinds of services, and mostly, handle behaviors governed by specific context information (typical characteristics of mobile applications). Our work follows an similar approach, creating an architecture able to handle issues like multiple asynchronous robot sensors and actuators and developing a set of artifacts specifically designed for exploiting ROS functionalities using JaCa.

10. CONCLUSION

In this work, we developed JaCaROS: an integration architecture between Jason - an agent programming language (APL) and ROS - a robot development framework (RDF), in which CArtAgO artifacts are used as the main abstraction for the robot sensors and actuators. It is composed by an interface responsible for translating APL higher-level abstraction into low-level robot control directives and translating data gathered from the environment into information that the agent can understand; and it supports the development of new applications for mobile robots, including the design and use of new sensors and actuators supported by the RDF.

The main contribution of this work to the area of mobile robot programming and multi-agent system programming is that we expanded Jason beyond the development of multi-agent systems, improving it to a high level mobile robot programming language. Our work simplifies the development of complex behaviors on robots in a practical, extensible and scalable way.

The development using JaCaROS is composed by two parts: the agent code developed in Jason, and the artifacts code developed in Java. This characteristic generates a disconnection between the agents layer and the interface layer, allowing a modular architecture and improving reuse and readability of robot programs. The simplification of code readability and code reuse is considered a contribution of this work: JaCaROS artifacts code and Jason code can be reused to fulfill same objectives in different projects, such as the use of the same code in a simulated environment and a real environment. Further, a JaCaROS code can be reused for different robots if they use the same ROS topic or just by adjusting the value of a couple of properties within the JaCaROS artifact.

Our architecture has some limitations regarding the artifact responsible for abstracting the environment into a Cartesian coordinate system. The current artifact supports only 2D positioning, therefore, only ground robots are able to use it. Improve this artifact in order to support 3D positioning is considered a future work. The same approach should be taken on the Movement artifact, adding move up and move down operations to it, and as a consequence allowing aerial robots, such as drones, to use the artifact. A second limitation is that our current architecture does not support the use of ROS actions, thus we are not able to use the native ROS path planners, transferring the responsibility to make a planner to JaCaROS developers. We consider a future work the integration of ROS actions with our architecture. Finally, as a future work, our integration should be implemented under other agent programming languages which support CArtAgO. We believe that the JaCaROS artifacts created to be used with Jason could also work with no changes in other APLs, since JaCaROS artifacts are designed in a modular fashion. Once our architecture does not change the basics artifact communication mechanisms (such as observable properties and operations), any APL able to use a CArtAgO artifact is also able to use a JaCaROS artifact.

BIBLIOGRAPHY

[1] Bellifemine, F. L.; Caire, G.; Greenwood, D. "Developing multi-agent systems with JADE". John Wiley & Sons, 2007, vol. 7, 300p.

[2] Berns, K.; von Puttkamer, E. "Autonomous Land Vehicles: Steps towards Service Robots". Springer-Verlag New York Incorporated, 2009, 283p.

[3] Bordini, R. H.; Braubach, L.; Dastani, M.; Gomez-Sanz, J. J.; Leite, J.; Pokahr, A.; Ricci, A. "A survey of programming languages and platforms for multi-agent systems", *Informatica (Slovenia)*, vol. 30, 2006, pp. 33–44.

[4] Bordini, R. H.; Hübner, J. F.; Wooldridge, M. "Jason: a java-based interpreter for a extended version of AgentSpeak(L)". Retrieved from: http://jason.sourceforge.net/wp/description/, November 2013.

[5] Bordini, R. H.; Hübner, J. F.; Wooldridge, M. "Programming multi-agent systems in AgentSpeak using Jason". John Wiley & Sons, 2007, vol. 8, 292p.

[6] Bratman, M. E. "Intention, plans, and practical reason". Harvard University Press, 1987, 200p.

[7] Busetta, P.; Rönnquist, R.; Hodgson, A.; Lucas, A. "JACK intelligent agents - components for intelligent agents in java", *Agentlink - European Co-ordination Action for Agent Based Computing*, vol. 2, 2005, pp. 2–5.

[8] Clocksin, W. F.; Mellish, C. S. "Programming in Prolog". Springer-Verlag, 1984, 2 ed., 297p.

[9] Dastani, M. "2APL: a practical agent programming language", *Autonomous Agents and Multi-Agent Systems*, vol. 16–3, 2008, pp. 214–248.

[10] de Nunes, I. O. "Implementação do modelo e da arquitetura BDI", *Monografias em Ciência da Computação*, vol. 1, 2007.

[11] D'Inverno, M.; Luck, M.; Georgeff, M.; Kinny, D.; Wooldridge, M. "The dMARS architecture: A specification of the distributed multi-agent reasoning system", *Autonomous Agents and Multi-Agent Systems*, vol. 9–1-2, 2004, pp. 5–53.

[12] Dudek, G.; Jenkin, M. "Computational Principles of Mobile Robotics". Cambridge University Press, 2010, 2 ed., 391p.

[13] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1994, 416p.

[14] Génération Robots. "Génération robots website". Retrieved from: http://www.generationrobots.com/, April 2014.

[15] Gerkey, B.; Vaughan, R. T.; Howard, A. “The player/stage project: Tools for multi-robot and distributed sensor systems”. In: Proceedings of the 11th international conference on advanced robotics, 2003, pp. 317–323.

[16] Goebel, P. “ROS By Example - A Do-It-Yourself Guide to the Robot Operating System”. Lulu, 2013, vol. 1, 467p.

[17] Goebel, P. “ROS By Example - Packages and Programs for Advanced Robot Behaviors”. Lulu, 2014, vol. 2, 272p.

[18] Haugeland, J. “Artificial Intelligence: The Very Idea”. A Bradford Book, 1989, 299p.

[19] Henning, M. “A new approach to object-oriented middleware”. In: *IEEE Internet Computing*, IEEE, 2004, vol. 8, pp. 66–75.

[20] Hibbeler, R. C. “Engineering Mechanics : Statics & Dynamics”. LinkPrentice-Hall, 1995, 7 ed., 624p.

[21] Hindrijs, K. V.; de Boer, F. S.; van der Hoek, W.; Meyer, J.-J. C. “Agent programming in 3APL”. In: Autonomous Agents and Multi-Agent Systems, 1999, pp. 357–401.

[22] Irish, J. “Lecture on instrumentation specifications”. Course 13.998, 2005.

[23] Kan, S. H. “Metrics and Models in Software Quality Engineering”. Boston, MA, USA: Addison-Wesley Longman, 2003, 2nd ed., 528p.

[24] Konolige, K. “A gradient method for realtime robot control”. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2000, pp. 639–646.

[25] Lars Braubach, A. P.; Lamersdorf, W. “Jadex: A BDI reasoning engine”. In: Multi-Agent Programming: Languages, Platforms and Applications, 2005, pp. 149–174.

[26] Lister, M.; Salem, T. “Design and implementation of a robot power supply system”. In: Proceedings of IEEE SoutheastCon, 2002, pp. 418–421.

[27] Machado, R.; Bordini, R. H. “Running AgentSpeak(L) agents on SIM AGENT”. In: Pre-Proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages, 2001, pp. 158–174.

[28] Makarenko, A.; Brooks, A.; Kaupp, T. “Orca: Components for robotics”. In: Proceedings of International Conference on Intelligent Robots and Systems, 2006, pp. 163–168.

[29] Montemerlo, D.; Roy, N.; Thrun, S. “Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (CARMEN) toolkit”. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003, pp. 2436–2441.

[30] Newman, P. M. "MOOS - mission orientated operating suite", *MIT Department of Ocean Engineering*, vol. 1, 2003, technical report OE2003-07.

[31] Newton, I. "Philosophiae Naturalis Principia Mathematica". J. Societatis Regiae ac Typis J. Streater, 1687, 680p.

[32] Nilsson, N. J. "Artificial Intelligence: A New Synthesis". Morgan Kaufmann, 1998, 513p.

[33] Player Project. "Gazebo: 3D multiple robot simulator with dynamics". Retrieved from: http://gazebosim. org, April 2014.

[34] Quigley, M.; Gerkey, B.; Conley, K.; Faust, J.; Foote, T.; Leibs, J.; Berger, E.; Wheeler, R.; Ng, A. "ROS: an open-source robot operating system". In: ICRA workshop on open source software, 2009.

[35] Rao, A. S. "AgentSpeak(L): BDI agents speak out in a logical computable language". In: Proceedings of the 7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, 1996, pp. 42–55.

[36] Rao, A. S.; Georgeff, M.; Ingrand, F. "An architecture for real-time reasoning and system control". In: Proceedings of the 7th European workshop on Modeling autonomous agents in a multi-agent world : agents breaking away: agents breaking away, 1992, pp. 34–44.

[37] Rao, A. S.; Georgeff, M. P.; et al.. "BDI agents: From theory to practice". In: Proceedings of the First International Conference on Multi-Agent Systems, 1995, pp. 312–319.

[38] Ricci, A.; Piunti, M.; Viroli, M. "Environment programming in multi-agent systems: an artifact-based perspective", *Autonomous Agents and Multi-Agent Systems*, vol. 23–2, 2011, pp. 158–192.

[39] Ricci, A.; Piunti, M.; Viroli, M.; Omicini, A. "Environment programming in CArtAgO". In: *Multi-Agent Programming: Languages, Tools and Applications*, Springer US, 2009, pp. 259–288.

[40] Ricci, A.; Viroli, M.; Omicini, A. "CArtAgO: An infrastructure for engineering computational environments in MAS", *Environments for Multi-Agent Systems*, vol. 1, 2006, pp. 102–119.

[41] Russell, S.; Norvig, P. "Artificial Intelligence: A Modern Approach". Prentice Hall, 2010, 3 ed., 1132p.

[42] Santi, A.; Guidi, M.; Ricci, A. "JaCa-Android: An agent-based platform for building smart mobile applications". In: Languages, Methodologies, and Development Tools for Multi-Agent Systems, 2010, pp. 95–114.

[43] Shoham, Y. "Agent oriented programming: An overview of the framework and summary of recent research". In: *Knowledge Representation and Reasoning Under Uncertainty*, Bradshaw, J. M. (Editor), Cambridge, MA, USA: Springer Berlin Heidelberg, 1994, vol. 808, pp. 123–129.

[44] Siciliano, B.; Sciavicco, L.; Villani, L.; Oriolo, G. “Robotics: Modelling, Planning and Control”. Springer Science & Business Media, 2009, 632p.

[45] Siegwart, R.; Nourbakhsh, I. R. “Introduction to Autonomous Mobile Robots”. The MIT Press, 2004, 453p.

[46] Simmons, R. “The inter-process communication (IPC) system”. Retrieved from: http://www.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html, March 2013.

[47] Stanford Artificial Intelligence Laboratory. “ROS groovy galapagos website”. Retrieved from: http://wiki.ros.org/, April 2013.

[48] Vaughan, R. T.; Gerkey, B. P.; Howard, A. “On device abstractions for portable, reusable robot code”. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003, pp. 2121–2427.

[49] Verbeek, M. “3APL as programming language for cognitive robots”, Master’s Thesis, Utrecht University, 2003.

[50] Weiss, G. “Multiagent Systems”. Massachusetts Institute of Technology Press, 2013, 867p.

[51] Wesz, R.; Meneguzzi, F. “Integrating robot control into the AgentSpeak(L) programming language”. In: Proceedings of the Eighth Workshop-School Agent Systems, Their Environments and Applications, 2014, pp. 197–203.

[52] Willow Garage. “Willow garage website”. Retrieved from: https://www.willowgarage.com, April 2014.

[53] Winston, P. H. “Artificial Intelligence”. Addison-Wesley, 1992, 3 ed., 737p.

[54] Wooldridge, M. “Michael wooldridge site at university of oxford”. Retrieved from: http://www.cs.ox.ac.uk/people/michael.wooldridge/pubs/imas/distrib/powerpoint-slides/, July 2013.

[55] Wooldridge, M. “An Introduction to MultiAgent Systems”. John Wiley & Sons, 2009, 461p.

[56] Wooldridge, M.; Jennings, N. R. “Intelligent agents: Theory and practice”, *The Knowledge Engineering Review*, vol. 10–02, 1995, pp. 115–152.

[57] Ziafati, P. “Programming autonomous robots using agent programming languages”. In: Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2013, pp. 1463–1464.

APPENDIX A – JASON CODE

```
+!create_odom(AgentName) :
    <- .print("Creating Odometry artifact. We are using an agent name: ", AgentName);
       makeArtifact("Odom","jason.architecture.ArtOdometry",[AgentName],OdomId).

-!create_odom(AgentName) :
    <- .print("Odometry artifact creation failed.").

+!create_odom :
    <- .print("Creating Odometry artifact. The agent name is unknown.");
       makeArtifact("Odom","jason.architecture.ArtOdometry",[],OdomId).

-!create_odom :
    <- .print("Odometry artifact creation failed.").

+?toolOdom(OdomId) :
    <- .print("Discovering Odometry artifact.");
       lookupArtifact("Odom",OdomId).

-?toolOdom(OdomId) :
    <- .print("Trying to discovering Odometry artifact again.");
       .wait(10);
       ?toolOdom(OdomId).

+odom(A,B,C)
    <- .print("An odometry property was perceived");
      println("X: ", A); println("Y: ", B); println("Z: ", C);
      -+currentPose(A,B,C).
```

Listing 26: Jason plans to create and use a JaCaROS artifact (ArtOdometry artifact was used as example).

APPENDIX B – ARTIFACTS CODE

We developed a JaCaROS artifact in which we abstract the environment into a Cartesian coordinate system. The artifact stores the information using a hashtable where the key is the concatenation of x axis and y axes ("X_Y") and the value are labels, such as: free, obstacle, home and goal. The *getVisionRange* operation updates the belief base with the data relative to a given coordinate position, avoiding the agent to update its belief base with all the hashtable content, as shown in Listing APPENDIX B.1. Therefore, instead of receive information of all the environment (saved within the artifact, in the hashtable), the agent receives partial information of the environment, in a radius around it.

```java
@OPERATION void getVisionRange(int x, int y, int range) {
  ObsProperty prop = getObsProperty("currentRange");

  for (int a=x-range; a<=x+range; a++) {
    for(int b=y-range; b<=y+range; b++) {

      if (slam_data.containsKey(a+"_"+b)) {
        position=slam_data.get(a+"_"+b);
        prop.updateValues(a,b,position);
        signal("currentRange");
      }
      else {
        System.out.println(a+"_"+b + " key not found!");
        }
      }
    }
  }
```

Listing APPENDIX B.1 – getVisionRange method from SLAM artifact.

```java
import cartago.*;
import geometry_msgs.*;
import org.ros.message.MessageListener;
import org.ros.node.topic.Subscriber;

public class ArtOdometry extends JaCaRosArtifact {
  private static String rosNodeName = "ArtOdometry";

  private Subscriber <nav_msgs.Odometry> subscriberOdometry;
  private String topicName = "/odom";
  private String topicType = nav_msgs.Odometry._TYPE;
  private String propertyName = "odom";

  private Point currentOdometry;
  private final byte refreshRateValue = 1; //must be > 0
  private byte refreshRateAux = refreshRateValue;

  private ReadCmd cmd;

```

```java
public ArtOdometry() {
  super(rosNodeName);
}

void init(String agentName) {
  defineObsProperty(propertyName, 0,0,0);

  if (agentName != null)
    topicName = "/" + agentName + topicName; // Update topic name

    cmd = new ReadCmd();
    subscriberOdometry = (Subscriber <nav_msgs.Odometry>) createSubscriber(topicName,
        topicType);
    subscriberOdometry.addMessageListener(new MessageListener <nav_msgs.Odometry> () {
      @Override
      public void onNewMessage(nav_msgs.Odometry message) {
        Point localPose = message.getPose().getPose().getPosition();

        if (currentOdometry == null)
          currentOdometry = localPose;
        else if (!currentOdometry.equals(localPose) && refreshRate()) {
          currentOdometry = localPose;
          execInternalOp("receiving");
          }
        }

      private boolean refreshRate() {
        boolean result = false;
        if (refreshRateAux % refreshRateValue == 0) {
          refreshRateAux = refreshRateValue;
          result = true;
        }

      refreshRateAux++;
      return result;
    }
  });
}

void init() {
  init(null);
}

@INTERNAL_OPERATION
void receiving() {
  await(cmd);
  signal(propertyName);
}

class ReadCmd implements IBlockingCmd {
  public ReadCmd() {}

```

```java
    public void exec() {
      try {
        ObsProperty prop = getObsProperty(propertyName);
        prop.updateValues(currentOdometry.getX(),currentOdometry.getY(),currentOdometry.getZ
            ());
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }

  @OPERATION void getCurrentOdom() {
    ObsProperty prop = getObsProperty(propertyName);
    if(currentOdometry!= null)
      prop.updateValues(currentOdometry.getX(),currentOdometry.getY(),currentOdometry.getZ()
          );
    else
      prop.updateValues(0,0,0);

    signal(propertyName);
  }
}
```

Listing APPENDIX B.2 – ArtOdometry source code.

```java
import cartago.*;
import geometry_msgs.*;
import org.ros.message.MessageListener;
import org.ros.node.topic.Subscriber;

public class ArtOdometryOnDemand extends JaCaRosArtifact {
  private static String rosNodeName = "ArtOdometry";

  Subscriber <nav_msgs.Odometry> subscriberOdometry;
  private String topicName = "/odom";
  private String topicType = nav_msgs.Odometry._TYPE;
  private String propertyName = "odom";

  private Point currentOdometry;

  public ArtOdometryOnDemand() {
    super(rosNodeName);
  }

  void init(String agentName) {
    defineObsProperty(propertyName, 0,0,0);
    if (agentName != null)
      topicName = "/" + agentName + topicName; // Update topic name

      subscriberOdometry = (Subscriber <nav_msgs.Odometry>) createSubscriber(topicName,
          topicType);
      subscriberOdometry.addMessageListener(new MessageListener <nav_msgs.Odometry> () {
      @Override
    public void onNewMessage(nav_msgs.Odometry message) {
      Point localPose = message.getPose().getPose().getPosition();
      if (currentOdometry == null)
        currentOdometry = localPose;
      else if (!currentOdometry.equals(localPose) )
        currentOdometry = localPose;
    }
  });
}

void init() {
  init(null);
}

@OPERATION void getCurrentOdom() {
  ObsProperty prop = getObsProperty(propertyName);
  if(currentOdometry!= null)
    prop.updateValues(currentOdometry.getX(),currentOdometry.getY(),currentOdometry.getZ()
        );
  else
    prop.updateValues(0,0,0);
  signal(propertyName);
}
}
```

Listing APPENDIX B.3 – ArtOdometryOnDemand source code.

```
import org.ros.exception.RemoteException;
import org.ros.exception.ServiceNotFoundException;
import org.ros.message.MessageListener;
import org.ros.node.service.ServiceClient;
import org.ros.node.service.ServiceResponseListener;
import org.ros.node.topic.Subscriber;
import test_rosjava_jni.AddTwoInts;
import test_rosjava_jni.AddTwoIntsRequest;
import test_rosjava_jni.AddTwoIntsResponse;
import cartago.IBlockingCmd;
import cartago.INTERNAL_OPERATION;
import cartago.OPERATION;
import cartago.ObsProperty;

public class ArtTutorial extends JaCaRosArtifact {
  private static String rosNodeName = "ArtTutorial";

  /* service properties */
  /** Property responsible to connects with AddTwoInts service */
  private ServiceClient<AddTwoIntsRequest, AddTwoIntsResponse> serviceClient;
  private long currentSum;
  /** Property to be observable within artifact */
  private String propertyNameSum = "twoIntsSum";
  private ReadCmdSum cmdSum;

  /* chatter properties */
```

```
/** Property responsible to connects with chatter topic */
private Subscriber<std_msgs.String> subscriberChatter;
private String topicName = "/chatter";
private String topicType = std_msgs.String._TYPE;
private String currentMessage;
/** Property to be observable within artifact */
private String propertyNameChatter = "chatter";
private ReadCmdMsg cmdMsg;

public ArtTutorial() {
  super(rosNodeName);
}

void init(String agentName) {
  if (agentName != null) {
    topicName = "/" + agentName + topicName; // Update topic name with agent name
    propertyNameChatter = agentName + "_" + propertyNameChatter;
    propertyNameSum = agentName + "_" + propertyNameSum;
  }

  defineObsProperty(propertyNameChatter, "");
  cmdMsg = new ReadCmdMsg();
  defineObsProperty(propertyNameSum, 0);
  cmdSum = new ReadCmdSum();

  try {
    serviceClient = (ServiceClient<AddTwoIntsRequest, AddTwoIntsResponse>) createClient("
        add_two_ints", AddTwoInts._TYPE);
  } catch (ServiceNotFoundException e) {
    e.printStackTrace();
  }

  subscriberChatter = (Subscriber <std_msgs.String>) createSubscriber(topicName, topicType
      );
  subscriberChatter.addMessageListener(new MessageListener <std_msgs.String> () {
  @Override
  public void onNewMessage(std_msgs.String message) {
    currentMessage = message.getData();
    execInternalOp("receivingMsg");
    }
  });
}

void init() {
  init(null);
}

@OPERATION void getCurrentMessage() {
  ObsProperty prop = getObsProperty(propertyNameChatter);
  if(currentMessage!= null)
    prop.updateValues(currentMessage);
  else
```

```
      prop.updateValues("Error: current message is null");

    signal(propertyNameChatter);
  }

  @OPERATION void sum(int valueA, int valueB){
    final AddTwoIntsRequest request = serviceClient.newMessage();
    request.setA(valueA);
    request.setB(valueB);
    serviceClient.call(request, new ServiceResponseListener<AddTwoIntsResponse>() {
      @Override
      public void onSuccess(AddTwoIntsResponse response) {
        currentSum = response.getSum();
        execInternalOp("receivingSum");
      }

      @Override
      public void onFailure(RemoteException e) {
        currentSum = 0;
      }
    });
  }

  @INTERNAL_OPERATION
  void receivingMsg() {
    await(cmdMsg);
    signal(propertyNameChatter);
  }

  class ReadCmdMsg implements IBlockingCmd {
    public void exec() {
    try {
      ObsProperty prop = getObsProperty(propertyNameChatter);
      prop.updateValues(currentMessage);
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }

  @INTERNAL_OPERATION
  void receivingSum() {
    await(cmdSum);
    signal(propertyNameSum);
  }

  class ReadCmdSum implements IBlockingCmd {
    public void exec() {
      try {
        ObsProperty prop = getObsProperty(propertyNameSum);
        prop.updateValues(currentSum);
      } catch (Exception ex) {
```

```
      ex.printStackTrace();
    }
  }
 }
}
```

Listing APPENDIX B.4 – ArtTutorial source code.

```
import cartago.*;
import geometry_msgs.*;
import java.util.List;
import java.util.ArrayList;
import org.ros.message.MessageListener;
import org.ros.node.topic.Publisher;
import org.ros.node.topic.Subscriber;

public class ArtCmdVelAndOdometry extends JaCaRosArtifact {
  private static String rosNodeName = "ArtCmdVelAndOdometry";

  /* cmd_vel properties */
  private Publisher<geometry_msgs.Twist> publisherCmdVel;
  private String topicNameCmdVel = "/cmd_vel";
  private String topicTypeCmdVel = geometry_msgs.Twist._TYPE;
  private String propertyNameCmdVel = "cmd_vel";
  private geometry_msgs.Twist twist;
  private float speed = 1;
  private double stepCounter = 0;
  private boolean abort = false;

  /* odom properties */
  Subscriber <nav_msgs.Odometry> subscriberOdometry;
  private String topicNameOdom = "/odom";
  private String topicTypeOdom = nav_msgs.Odometry._TYPE;
  private String propertyNameOdom = "odom_cmd_vel";
  private Point currentOdometry;

  public ArtCmdVelAndOdometry() {
    super(rosNodeName);
  }

  void init(String agentName) {
    if (agentName != null) { // Update topic name with agent name
      topicNameOdom = "/" + agentName + topicNameOdom;
      topicNameCmdVel = "/" + agentName + topicNameCmdVel;
      propertyNameOdom = agentName + "_" + propertyNameOdom;
    }

    /* cmd_vel communication */
    publisherCmdVel = (Publisher<Twist>) createPublisher(topicNameCmdVel, topicTypeCmdVel);
    sleep(2000);

    /* odom communication */
    defineObsProperty(propertyNameOdom, 0,0,0);

    subscriberOdometry = (Subscriber <nav_msgs.Odometry>) createSubscriber(topicNameOdom,
        topicTypeOdom);
    subscriberOdometry.addMessageListener(new MessageListener <nav_msgs.Odometry> () {
    @Override
    public void onNewMessage(nav_msgs.Odometry message) {
      Point localPose = message.getPose().getPose().getPosition();
      if (currentOdometry == null || !currentOdometry.equals(message.getPose().getPose().
          getPosition()))
        currentOdometry = localPose;

      }
    });
  }

  void init() {
    init(null);
  }

  /* cmd_vel Operations */

  @OPERATION void moveForward(double distance){
    twist = publisherCmdVel.newMessage();
    twist.getLinear().setX(this.speed); twist.getLinear().setY(0); twist.getLinear().setZ(0)
        ;
    twist.getAngular().setX(0); twist.getAngular().setY(0); twist.getAngular().setZ(0);

    this.abort = false; //initialize abort flag
    double initialPose;
    if (currentOdometry!=null)
      initialPose = currentOdometry.getX();
    else
      initialPose = 0;

    stepCounter = 0; //initialize stepcounter
    while (!this.abort && stepCounter <= distance)
    {
      if (currentOdometry!=null)
        stepCounter = currentOdometry.getX() - initialPose;

      sleepNoLog(10);
      publisherCmdVel.publish(twist);
    }
  }

  /* odom Operations */

  @OPERATION void abortMoving() {
    this.abort = true;
  }

  @OPERATION void rotate(int degree){
    twist = publisherCmdVel.newMessage();
    float localSpeed = 0.5f;
    if (degree < 0)
      {
      localSpeed = localSpeed * -1;
      degree = degree * -1;
      }
    twist.getLinear().setX(0); twist.getLinear().setY(0); twist.getLinear().setZ(0);
    twist.getAngular().setX(0); twist.getAngular().setY(0); twist.getAngular().setZ(
        localSpeed);

    for (short i=0; i<degree/15; i++)
      { //magic number depends of robot calibration
      publisherCmdVel.publish(twist);
      sleepNoLog(500);
      }
    }

  @OPERATION void getCurrentOdom() {
    ObsProperty prop = getObsProperty(propertyNameOdom);
    if(currentOdometry!= null)
      prop.updateValues(currentOdometry.getX(),currentOdometry.getY(),currentOdometry.getZ()
          );
    else
      prop.updateValues(0,0,0);
      signal(propertyNameOdom);
  }
}
```

Listing APPENDIX B.5 – ArtCmdVelAndOdometry source code.

```
public class ArtCmdVel extends JaCaRosArtifact {
  private Publisher<geometry_msgs.Twist> publisherCmdVel;
  private geometry_msgs.Twist twist = publisherCmdVel.newMessage();
  private static String rosNodeName = "ArtCmdVel";
  private String topicName = "/cmd_vel";
  private String topicType = geometry_msgs.Twist._TYPE;
  private float speed = 1;

  public ArtCmdVel(){
    super(rosNodeName); //Creates JaCaROS node into ROS
  }

  void init(){
    init(null);
    }

  void init(String agentName){
    if (agentName != null) // Update topic name with agent name
            topicName = "/" + agentName + topicName;

        publisherCmdVel = (Publisher<Twist>) createPublisher(topicName, topicType);
        sleep(2000); //take some time to ROS process last command
  }

@OPERATION void move(int linear_duration){
  twist.getLinear().setX(speed); twist.getLinear().setY(0); twist.getLinear().setZ(0);
  twist.getAngular().setX(0); twist.getAngular().setY(0); twist.getAngular().setZ(0);

  for (int a=0; a<=linear_duration; a++){ //publish during a given time
    publisherCmdVel.publish(twist);
    sleepNoLog(100);
  }
}

@OPERATION void rotate(int degree){
  if (degree < 0)
    {
    speed = speed * -1;
    degree = degree * -1;
    }

  twist.getLinear().setX(0); twist.getLinear().setY(0); twist.getLinear().setZ(0);
  twist.getAngular().setX(0); twist.getAngular().setY(0); twist.getAngular().setZ(speed);

  for (short i=0; i<degree/15; i++) { //magic number depends of robot calibration
    publisherCmdVel.publish(twist);
    sleepNoLog(100);
    }
}
}
```

Listing APPENDIX B.6 – ArtCmdVel source code.

```
public class ArtCmdVelMuxAndOdometry extends JaCaRosArtifact {
    /* cmd_vel properties */
    private Publisher<geometry_msgs.Twist> publisherCmdVelMux;
    private geometry_msgs.Twist twist = publisherCmdVelMux.newMessage();
    private static String rosNodeName = "ArtCmdVelMuxAndOdometry";
    private String propertyNameCmdVelMux = "cmd_vel_mux";
    private String topicNameCmdVelMux = "/cmd_vel_mux/input/navi";
    private String topicTypeCmdVelMux = geometry_msgs.Twist._TYPE;
    private float speed = 1;
    private double stepCounter = 0;
    private boolean abort = false;
    /* odom properties */
    Subscriber <nav_msgs.Odometry> subscriberOdometry;
    Point currentOdometry;
    private String propertyNameOdom = "odom_cmd_vel_mux";
    private String topicNameOdom = "/odom";
    private String topicTypeOdom = nav_msgs.Odometry._TYPE;

    public ArtCmdVelMuxAndOdometry() {
        super(rosNodeName); //Creates JaCaROS node into ROS
    }

    void init(String agentName) {
        if (agentName != null) {
            // Update topic name and property name with agent name (for multi-agent systems)
            topicNameOdom = "/" + agentName + topicNameOdom;
            topicNameCmdVelMux = "/" + agentName + topicNameCmdVelMux;
            propertyNameOdom = agentName + "_" + propertyNameOdom;
        }

        /* cmd_vel_mux communication */
        publisherCmdVelMux = (Publisher<Twist>) createPublisher(topicNameCmdVelMux,
            topicTypeCmdVelMux);
        sleep(2000); //take some time to ROS process last command
        /* odom communication */
        defineObsProperty(propertyNameOdom, 0,0,0);
        subscriberOdometry = (Subscriber <nav_msgs.Odometry>) createSubscriber(topicNameOdom
            , topicTypeOdom);
        /* odom Listener */
        subscriberOdometry.addMessageListener(new MessageListener <nav_msgs.Odometry> () {
            @Override
            public void onNewMessage(nav_msgs.Odometry message) {
                Point localPose = message.getPose().getPose().getPosition();
                if (currentOdometry == null || !currentOdometry.equals(message.getPose().
                    getPose().getPosition()))
                    currentOdometry = localPose;
            }
        });
    }

    void init() {
        init(null);
    }

    /* cmd_vel_mux Operations */
    @OPERATION void moveForward(double distance){
        twist.getLinear().setX(speed); twist.getLinear().setY(0); twist.getLinear().setZ(0);
        twist.getAngular().setX(0); twist.getAngular().setY(0); twist.getAngular().setZ(0);
        this.abort = false; //initialize abort flag
        double initialPose;
        if (currentOdometry!=null)
            initialPose = currentOdometry.getX();
        else
            initialPose = 0;

        stepCounter = 0; //initialize stepcounter
        while (!this.abort && stepCounter <= distance) {
            if (currentOdometry!=null)
                stepCounter = currentOdometry.getX() - initialPose;

            sleepNoLog(10);
            publisherCmdVelMux.publish(twist); //Publish into ROS topics: cmd_vel Linear X
        }

        if (this.abort)
          logger1.info("ArtCmdVelMuxAndOdometry >> ABORTED");
    }

    @OPERATION void rotate(int degree) {
        float localSpeed = this.speed;

        if (degree < 0) {
            localSpeed = localSpeed * -1;
            degree = degree * -1;
            }

        twist.getLinear().setX(0); twist.getLinear().setY(0); twist.getLinear().setZ(0);
        twist.getAngular().setX(0); twist.getAngular().setY(0); twist.getAngular().setZ(
            localSpeed);

        for (short i=0; i<degree/15; i++) { //magic number depends of robot calibration
                publisherCmdVelMux.publish(twist); //Publish into ROS topics: cmd_vel_mux Angular Z
                sleepNoLog(10);
            }
        }

    @OPERATION void abortMoving() {
        this.abort = true;
    }

    /* odom Operations */
    @OPERATION void getCurrentOdom() {
        ObsProperty prop = getObsProperty(propertyNameOdom);
        if(currentOdometry!= null)
            prop.updateValues(currentOdometry.getX(),currentOdometry.getY(),currentOdometry.
                getZ());
        else
            prop.updateValues(0,0,0);

        signal(propertyNameOdom);
    }
}
```

Listing APPENDIX B.7 – ArtCmdVelMuxAndOdometry artifact source code.

```
public class ArtSimBattery extends Artifact {
  private String propertyNameBattery = "battery";
  private int currentBattery = 100; //initialize battery 100% charged
  private String propertyNameDiagnostics = "diagnostics";
  private String currentDiagnostics = "Full Battery"; //initialize status = full charged
  private ReadCmd cmd;
  private Timer timer;
  private final int FULL = 100;
  private final int MEDIUM = 50;
  private final int LOW = 20;
  private final int STEP = 1;

```

```java
void init() {
  defineObsProperty(propertyNameBattery, currentBattery);
  signal(propertyNameBattery);
  defineObsProperty(propertyNameDiagnostics, currentDiagnostics);
  signal(propertyNameDiagnostics);

  cmd = new ReadCmd();
  timer = new Timer(1000, new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
      currentBattery = currentBattery - STEP; // uncharging
      execInternalOp("intOp");
      if (currentBattery == 0)
        timer.stop();
      }
  });
  timer.start();
}

@INTERNAL_OPERATION
void intOp() {
  await(cmd);
}

class ReadCmd implements IBlockingCmd {
  public ReadCmd() {}

  public void exec() {
    try {
      ObsProperty prop = getObsProperty(propertyNameBattery);
      prop.updateValues(currentBattery);
      isDiagnosticUpdateNeeded();
    }catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}

private void isDiagnosticUpdateNeeded() {
  String localDiagnostic = getObsProperty(propertyNameDiagnostics).stringValue(); //saving
      current
  switch (currentBattery) {
    case FULL:
      currentDiagnostics = "Full Battery";
      break;
    case MEDIUM:
      currentDiagnostics = "Medium Battery";
      break;
    case LOW:
      currentDiagnostics = "Low Battery";
      break;
  }
```

```
    if (!localDiagnostic.equals(currentDiagnostics)){
      ObsProperty propDiag = getObsProperty(propertyNameDiagnostics);
      propDiag.updateValues(currentDiagnostics);
    }
  }

  @OPERATION void operationRecharge() throws InterruptedException{
    while (currentBattery < 100) {
      currentBattery = currentBattery + 10;
      ObsProperty prop = getObsProperty(propertyNameBattery); //recharging
      prop.updateValues(currentBattery);
      signal(propertyNameBattery);
      Thread.sleep(500);
      isDiagnosticUpdateNeeded();
      }
    timer.start();
    }
}
```

Listing APPENDIX B.8 – ArtSimBattery pseudo code.

```
public class ArtCleaning extends Artifact {
  private String propertyNameAtRoom = "atRoom";
  private String propertyNameVacuum = "vacuum";
  private String propertyNameMop = "mop";
  private String propertyNameScrub = "scrub";

  private final String HALLWAY ="hallway";
  private final String LIVING ="living Room";
  private final String KITCHEN ="kitchen";
  private final String BATHROOM ="bathroom";

  private int atRoom = 0;
  private String room = HALLWAY;

  void init() {
    defineObsProperty(propertyNameAtRoom, atRoom); //initialize at hallway
    signal(propertyNameAtRoom);
    defineObsProperty(propertyNameVacuum, "", "");
    defineObsProperty(propertyNameMop, "", "");
    defineObsProperty(propertyNameScrub, "", "");
  }


  @OPERATION void whereAmI() {
    ObsProperty prop = getObsProperty(propertyNameAtRoom);
    signal(propertyNameAtRoom);
  }

  @OPERATION void clean(int localRoom) throws InterruptedException{
    if(localRoom == 0){
      room = HALLWAY;
    }else if (localRoom == 1){
      room = LIVING;
      ObsProperty prop = getObsProperty(propertyNameVacuum);
      prop.updateValues("Carpet", room);
      signal(propertyNameVacuum);
    }else if (localRoom == 2){
      room = KITCHEN;
      ObsProperty prop = getObsProperty(propertyNameMop);
      prop.updateValues("Flor", room);
      signal(propertyNameMop);
    }else if (localRoom == 3){
      room = BATHROOM;
      ObsProperty prop = getObsProperty(propertyNameScrub);
      prop.updateValues("Tub", room);
      signal(propertyNameScrub);

      Thread.sleep(1000);
      prop = getObsProperty(propertyNameMop);
      prop.updateValues("Flor", room);
      signal(propertyNameMop);
    }else {
      room = "error";
    }
    Thread.sleep(2500);
  }

  @OPERATION void updateWhereAmI(){
    atRoom++;
    if (atRoom > 3)
      atRoom = 0;

    ObsProperty prop = getObsProperty(propertyNameAtRoom);
    prop.updateValues(atRoom);
    signal(propertyNameAtRoom);
  }
}
```

Listing APPENDIX B.9 – ArtCleaning source code.

APPENDIX C – PYTHON CODE

```python
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from math import pi

class OutAndBack():
    def __init__(self):
        rospy.init_node('out_and_back', anonymous=False) # Give the node a name
        rospy.on_shutdown(self.shutdown) # Set rospy to execute a shutdown function when exiting
        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist) # Publisher to control the robot's speed
        rate = 50 # How fast will we update the robot's movement?
        r = rospy.Rate(rate) # Set the equivalent ROS rate variable
        linear_speed = 0.2 # Set the forward linear speed to 0.2 meters per second
        goal_distance = 1.0 # Set the travel distance to 1.0 meters
        linear_duration = goal_distance / linear_speed # How long should it take us to get there?
        angular_speed = 1.0 # Set the rotation speed to 1.0 radians per second
        goal_angle = pi # Set the rotation angle to Pi radians (180 degrees)
        angular_duration = goal_angle / angular_speed # How long should it take to rotate?

        # Loop through the two legs of the trip
        for i in range(2):
            move_cmd = Twist() # Initialize the movement command
            move_cmd.linear.x = linear_speed # Set the forward speed
            ticks = int(linear_duration * rate) # Move forward for a time to go the desired distance

            for t in range(ticks):
                self.cmd_vel.publish(move_cmd)
                r.sleep()

            # Stop the robot before the rotation
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1)

            # Now rotate left roughly 180 degrees
            # Set the angular speed
            move_cmd.angular.z = angular_speed

            ticks = int(goal_angle * rate) # Rotate for a time to go 180 degrees

            for t in range(ticks):
                self.cmd_vel.publish(move_cmd)
                r.sleep()

            # Stop the robot before the next leg
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1)

        # Stop the robot
        self.cmd_vel.publish(Twist())

    def shutdown(self):
        # Always stop the robot when shutting down the node.
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        OutAndBack()
    except:
        rospy.loginfo("Out-and-Back node terminated.")
```

Listing APPENDIX C.1 – The Time-Based Out-and-Back Script [16].

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist, Point, Quaternion
import tf
from rbx1_nav.transform_utils import quat_to_angle, normalize_angle
from math import radians, copysign, sqrt, pow, pi

class NavSquare():
    def __init__(self):
        rospy.init_node('nav_square', anonymous=False) # Give the node a name
        rospy.on_shutdown(self.shutdown) # Set rospy to execute a shutdown function when terminating
            the script
        rate = 20 # How fast will we check the odometry values?
        r = rospy.Rate(rate) # Set the equivalent ROS rate variable

        # Set the parameters for the target square
        goal_distance = rospy.get_param("~goal_distance", 1.0) # meters
        goal_angle = rospy.get_param("~goal_angle", radians(90)) # degrees converted to radians
        linear_speed = rospy.get_param("~linear_speed", 0.2)  # meters per second
        angular_speed = rospy.get_param("~angular_speed", 0.7) # radians per second
        angular_tolerance = rospy.get_param("~angular_tolerance", radians(2)) # degrees to
            radians

        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist) # Publisher to control the robot's speed
        self.base_frame = rospy.get_param('~base_frame', '/base_link') # The base frame is
            base_footprint for the TurtleBot but base_link for Pi Robot
        self.odom_frame = rospy.get_param('~odom_frame', '/odom') # The odom frame is usually
            just /odom
        self.tf_listener = tf.TransformListener() # Initialize the tf listener
        rospy.sleep(2) # Give tf some time to fill its buffer
        self.odom_frame = '/odom' # Set the odom frame

        try:
            self.tf_listener.waitForTransform(self.odom_frame, '/base_footprint', rospy.Time
                (), rospy.Duration(1.0))
            self.base_frame = '/base_footprint'
```

```python
        except (tf.Exception, tf.ConnectivityException, tf.LookupException):
            try:
                self.tf_listener.waitForTransform(self.odom_frame, '/base_link', rospy.Time(), rospy.Duration(1.0))
                self.base_frame = '/base_link'
            except (tf.Exception, tf.ConnectivityException, tf.LookupException):
                rospy.loginfo("Cannot find transform between /odom and /base_link or /base_footprint")
                rospy.signal_shutdown("tf Exception")

        position = Point() # Initialize the position variable as a Point type

        # Cycle through the four sides of the square
        for i in range(4):
            move_cmd = Twist() # Initialize the movement command
            move_cmd.linear.x = linear_speed # Set the movement command to forward motion
            (position, rotation) = self.get_odom() # Get the starting position values

            x_start = position.x
            y_start = position.y

            distance = 0 # Keep track of the distance traveled

            # Enter the loop to move along a side
            while distance < goal_distance and not rospy.is_shutdown():
                # Publish the Twist message and sleep 1 cycle
                self.cmd_vel.publish(move_cmd)
                r.sleep()

                (position, rotation) = self.get_odom() # Get the current position

                # Compute the Euclidean distance from the start
                distance = sqrt(pow((position.x - x_start), 2) +
                              pow((position.y - y_start), 2))

            # Stop the robot before rotating
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1.0)

            move_cmd.angular.z = angular_speed # Set the movement command to a rotation
            last_angle = rotation # Track the last angle measured
            turn_angle = 0 # Track how far we have turned

            # Begin the rotation
            while abs(turn_angle + angular_tolerance) < abs(goal_angle) and not rospy.is_shutdown():
                self.cmd_vel.publish(move_cmd) # Publish the Twist message and sleep 1 cycle
                r.sleep()
                (position, rotation) = self.get_odom() # Get the current rotation
                delta_angle = normalize_angle(rotation - last_angle) # Compute the amount of rotation since the last loop

                turn_angle += delta_angle
                last_angle = rotation

            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1.0)

        self.cmd_vel.publish(Twist()) # Stop the robot when we are done

    def get_odom(self):
        # Get the current transform between the odom and base frames
        try:
            (trans, rot) = self.tf_listener.lookupTransform(self.odom_frame, self.base_frame, rospy.Time(0))
        except (tf.Exception, tf.ConnectivityException, tf.LookupException):
            rospy.loginfo("TF Exception")
            return

        return (Point(*trans), quat_to_angle(Quaternion(*rot)))

    def shutdown(self):
        # Always stop the robot when shutting down the node
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        NavSquare()
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation terminated.")
```

Listing APPENDIX C.2 – Navigating a Square [16].

```python
import rospy
from diagnostic_msgs.msg import *
from std_msgs.msg import Float32
from rbx2_msgs.srv import *
import dynamic_reconfigure.server
from rbx2_utils.cfg import BatterySimulatorConfig
import thread

class BatterySimulator():
  def __init__(self):
    rospy.init_node("battery_simulator")
    self.rate = rospy.get_param("~rate", 1) # The rate at which to publish the battery level
    r = rospy.Rate(self.rate) # Convert to a ROS rate
    self.battery_runtime = rospy.get_param("~battery_runtime", 30) # The battery runtime in seconds
    self.initial_battery_level = rospy.get_param("~initial_battery_level", 100) # The intial battery level - 100 is considered full charge
    self.error_battery_level = rospy.get_param("~error_battery_level", 20) # Error battery level for diagnostics
    self.warn_battery_level = rospy.get_param("~warn_battery_level", 50) # Warn battery level for diagnostics
    self.current_battery_level = self.initial_battery_level # Initialize the current level variable to the startup level
    self.new_battery_level = self.initial_battery_level # Initialize the new level variable to the startup level
    self.battery_step = float(self.initial_battery_level) / self.rate / self.battery_runtime # The step sized used to decrease the battery level on each publishing loop
    self.mutex = thread.allocate_lock() # Reserve a thread lock
    battery_level_pub = rospy.Publisher("battery_level", Float32) # Create the battery level publisher
    rospy.Service('~set_battery_level', SetBatteryLevel, self.SetBatteryLevelHandler) # A service to maually set the battery level
    diag_pub = rospy.Publisher("diagnostics", DiagnosticArray) # Create a diagnostics publisher
    dyn_server = dynamic_reconfigure.server.Server(BatterySimulatorConfig, self.dynamic_reconfigure_callback) # Create a dynamic_reconfigure server and set a callback function
    rospy.loginfo("Publishing simulated battery level with a runtime of " + str(self.battery_runtime) + " seconds...")

    # Start the publishing loop
    while not rospy.is_shutdown():
      # Initialize the diagnostics status
      status = DiagnosticStatus()
      status.name = "Battery Level"

      # Set the diagnostics status level based on the current battery level
      if self.current_battery_level < self.error_battery_level:
        status.message = "Low Battery"
        status.level = DiagnosticStatus.ERROR
      elif self.current_battery_level < self.warn_battery_level:
        status.message = "Medium Battery"
        status.level = DiagnosticStatus.WARN
      else:
        status.message = "Battery OK"
        status.level = DiagnosticStatus.OK

      status.values.append(KeyValue("Battery Level", str(self.current_battery_level))) # Add the raw battery level to the diagnostics message

      # Build the diagnostics array message
      msg = DiagnosticArray()
      msg.header.stamp = rospy.Time.now()
      msg.status.append(status)
      diag_pub.publish(msg)
      battery_level_pub.publish(self.current_battery_level)
      self.current_battery_level = max(0, self.current_battery_level - self.battery_step)
      r.sleep()

  def dynamic_reconfigure_callback(self, config, level):
    if self.battery_runtime != config['battery_runtime']:
      self.battery_runtime = config['battery_runtime']
      self.battery_step = 100.0 / self.rate / self.battery_runtime

    if self.new_battery_level != config['new_battery_level']:
      self.new_battery_level = config['new_battery_level']
      self.mutex.acquire()
      self.current_battery_level = self.new_battery_level
      self.mutex.release()

    return config

  def SetBatteryLevelHandler(self, req):
    self.mutex.acquire()
    self.current_battery_level = req.value
    self.mutex.release()
    return SetBatteryLevelResponse()

if __name__ == '__main__':
  BatterySimulator()
```

Listing APPENDIX C.3 – The Fake Battery Simulator Script [17].

```
import rospy
import actionlib
from actionlib import GoalStatus
from geometry_msgs.msg import Pose, Point, Quaternion, Twist
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal, MoveBaseActionFeedback
from tf.transformations import quaternion_from_euler
from visualization_msgs.msg import Marker
from math import pi
from collections import OrderedDict

def setup_task_environment(self):
  # How big is the square we want the robot to patrol?
  self.square_size = rospy.get_param("~square_size", 1.0) # meters
  # Set the low battery threshold (between 0 and 100)
  self.low_battery_threshold = rospy.get_param('~low_battery_threshold', 50)
  # How many times should we execute the patrol loop
  self.n_patrols = rospy.get_param("~n_patrols", 2) # meters
  # How long do we have to get to each waypoint?
  self.move_base_timeout = rospy.get_param("~move_base_timeout", 10) #seconds
  self.patrol_count = 0 # Initialize the patrol counter

  # Subscribe to the move_base action server
  self.move_base = actionlib.SimpleActionClient("move_base", MoveBaseAction)
  rospy.loginfo("Waiting for move_base action server...")
  # Wait up to 60 seconds for the action server to become available
  self.move_base.wait_for_server(rospy.Duration(60))
  rospy.loginfo("Connected to move_base action server")

  quaternions = list() # Create a list to hold the target quaternions (orientations)
  euler_angles = (pi/2, pi, 3*pi/2, 0) # First define the corner orientations as Euler angles
  # Then convert the angles to quaternions
  for angle in euler_angles:
    q_angle = quaternion_from_euler(0, 0, angle, axes='sxyz')
    q = Quaternion(*q_angle)
    quaternions.append(q)

  self.waypoints = list() # Create a list to hold the waypoint poses

  # Append each of the four waypoints to the list. Each waypoint
  # is a pose consisting of a position and orientation in the map frame.
  self.waypoints.append(Pose(Point(0.0, 0.0, 0.0), quaternions[3]))
  self.waypoints.append(Pose(Point(self.square_size, 0.0, 0.0), quaternions[0]))
  self.waypoints.append(Pose(Point(self.square_size, self.square_size, 0.0), quaternions[1]))
  self.waypoints.append(Pose(Point(0.0, self.square_size, 0.0), quaternions[2]))

  # Create a mapping of room names to waypoint locations
  room_locations = (('hallway', self.waypoints[0]),
           ('living_room', self.waypoints[1]),
           ('kitchen', self.waypoints[2]),
           ('bathroom', self.waypoints[3]))

  # Store the mapping as an ordered dictionary so we can visit the rooms in sequence
  self.room_locations = OrderedDict(room_locations)
  # Where is the docking station?
  self.docking_station_pose = (Pose(Point(0.5, 0.5, 0.0), Quaternion(0.0, 0.0, 0.0, 1.0)))
  init_waypoint_markers(self) # Initialize the waypoint visualization markers for RViz
  # Set a visualization marker at each waypoint
  for waypoint in self.waypoints:
    p = Point()
    p = waypoint.position
    self.waypoint_markers.points.append(p)
  init_docking_station_marker(self) # Set a marker for the docking station

  # Publisher to manually control the robot (e.g. to stop it)
  self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
  rospy.loginfo("Starting Tasks")
  # Publish the waypoint markers
  self.marker_pub.publish(self.waypoint_markers)
  rospy.sleep(1)
  self.marker_pub.publish(self.waypoint_markers)
  # Publish the docking station marker
  self.docking_station_marker_pub.publish(self.docking_station_marker)
  rospy.sleep(1)

def init_waypoint_markers(self):
  # Set up our waypoint markers
  marker_scale = 0.2
  marker_lifetime = 0 # 0 is forever
  marker_ns = 'waypoints'
  marker_id = 0
  marker_color = {'r': 1.0, 'g': 0.7, 'b': 1.0, 'a': 1.0}
  # Define a marker publisher.
```

```
    self.marker_pub = rospy.Publisher('waypoint_markers', Marker)
    # Initialize the marker points list.
    self.waypoint_markers = Marker()
    self.waypoint_markers.ns = marker_ns
    self.waypoint_markers.id = marker_id
    self.waypoint_markers.type = Marker.CUBE_LIST
    self.waypoint_markers.action = Marker.ADD
    self.waypoint_markers.lifetime = rospy.Duration(marker_lifetime)
    self.waypoint_markers.scale.x = marker_scale
    self.waypoint_markers.scale.y = marker_scale
    self.waypoint_markers.color.r = marker_color['r']
    self.waypoint_markers.color.g = marker_color['g']
    self.waypoint_markers.color.b = marker_color['b']
    self.waypoint_markers.color.a = marker_color['a']
    self.waypoint_markers.header.frame_id = 'odom'
    self.waypoint_markers.header.stamp = rospy.Time.now()
    self.waypoint_markers.points = list()

def init_docking_station_marker(self):
    # Define a marker for the charging station
    marker_scale = 0.3
    marker_lifetime = 0 # 0 is forever
    marker_ns = 'waypoints'
    marker_id = 0
    marker_color = {'r': 0.7, 'g': 0.7, 'b': 0.0, 'a': 1.0}
    self.docking_station_marker_pub = rospy.Publisher('docking_station_marker', Marker)
    self.docking_station_marker = Marker()
    self.docking_station_marker.ns = marker_ns
    self.docking_station_marker.id = marker_id
    self.docking_station_marker.type = Marker.CYLINDER
    self.docking_station_marker.action = Marker.ADD
    self.docking_station_marker.lifetime = rospy.Duration(marker_lifetime)
    self.docking_station_marker.scale.x = marker_scale
    self.docking_station_marker.scale.y = marker_scale
    self.docking_station_marker.scale.z = 0.02
    self.docking_station_marker.color.r = marker_color['r']
    self.docking_station_marker.color.g = marker_color['g']
    self.docking_station_marker.color.b = marker_color['b']
    self.docking_station_marker.color.a = marker_color['a']
    self.docking_station_marker.header.frame_id = 'odom'
    self.docking_station_marker.header.stamp = rospy.Time.now()
    self.docking_station_marker.pose = self.docking_station_pose
```

Listing APPENDIX C.4 – Task Setup script. It establish the basic environment [17].

```
#!/usr/bin/env python
import string
import pygraphviz as pgv
from pygraph.classes.graph import graph
from pygraph.classes.digraph import digraph
from pygraph.algorithms.searching import breadth_first_search
from pygraph.readwrite.dot import write

```

```
class TaskStatus(object):
  """ A class for enumerating task statuses """
  FAILURE = 0
  SUCCESS = 1
  RUNNING = 2

class Task(object):
  """ "The base Task class """
  def __init__(self, name, children=None, *args, **kwargs):
    self.name = name
    self.status = None
    if children is None:
      children = []
    self.children = children

  def run(self):
    pass

  def reset(self):
    for c in self.children:
      c.reset()

  def add_child(self, c):
    self.children.append(c)

  def remove_child(self, c):
    self.children.remove(c)

  def prepend_child(self, c):
    self.children.insert(0, c)

  def insert_child(self, c, i):
    self.children.insert(i, c)

  def get_status(self):
    return self.status

  def set_status(self, s):
    self.status = s

  def announce(self):
    print("Executing task " + str(self.name))

  # These next two functions allow us to use the 'with' syntax
  def __enter__(self):
    return self.name

  def __exit__(self, exc_type, exc_val, exc_tb):
    if  exc_type is not None:
      return False
    return True

```

```python
class Selector(Task):
  """ A selector runs each task in order until one succeeds,
    at which point it returns SUCCESS. If all tasks fail, a FAILURE
    status is returned. If a subtask is still RUNNING, then a RUNNING
    status is returned and processing continues until either SUCCESS
    or FAILURE is returned from the subtask.
  """
  def __init__(self, name, *args, **kwargs):
    super(Selector, self).__init__(name, *args, **kwargs)

  def run(self):
    for c in self.children:
      c.status = c.run()
      if c.status != TaskStatus.FAILURE:
        return c.status
    return TaskStatus.FAILURE

class Sequence(Task):
  """
    A sequence runs each task in order until one fails,
    at which point it returns FAILURE. If all tasks succeed, a SUCCESS
    status is returned. If a subtask is still RUNNING, then a RUNNING
    status is returned and processing continues until either SUCCESS
    or FAILURE is returned from the subtask.
  """
  def __init__(self, name, *args, **kwargs):
    super(Sequence, self).__init__(name, *args, **kwargs)

  def run(self):
    for c in self.children:
      c.status = c.run()
      if c.status != TaskStatus.SUCCESS:
        return c.status
    return TaskStatus.SUCCESS

class Iterator(Task):
  """
    Iterate through all child tasks ignoring failure.
  """
  def __init__(self, name, *args, **kwargs):
    super(Iterator, self).__init__(name, *args, **kwargs)

  def run(self):
    for c in self.children:
      c.status = c.run()
      if c.status != TaskStatus.SUCCESS and c.status != TaskStatus.FAILURE:
        return c.status
    return TaskStatus.SUCCESS

class ParallelOne(Task):
  """
    A parallel task runs each child task at (roughly) the same time.
    The ParallelOne task returns success as soon as any child succeeds.
  """
  def __init__(self, name, *args, **kwargs):
    super(ParallelOne, self).__init__(name, *args, **kwargs)

  def run(self):
    for c in self.children:
      c.status = c.run()
      if c.status == TaskStatus.SUCCESS:
        return TaskStatus.SUCCESS
    return TaskStatus.FAILURE

class ParallelAll(Task):
  """
    A parallel task runs each child task at (roughly) the same time.
    The ParallelAll task requires all subtasks to succeed for it to succeed.
  """
  def __init__(self, name, *args, **kwargs):
    super(ParallelAll, self).__init__(name, *args, **kwargs)

  def run(self):
    n_success = 0
    n_children = len(self.children)

    for c in self.children:
      c.status = c.run()
      if c.status == TaskStatus.SUCCESS:
        n_success += 1
      if c.status == TaskStatus.FAILURE:
        return TaskStatus.FAILURE
    if n_success == n_children:
      return TaskStatus.SUCCESS
    else:
      return TaskStatus.RUNNING

class Loop(Task):
  """
    Loop over one or more subtasks for the given number of iterations
    Use the value -1 to indicate a continual loop.
  """
  def __init__(self, name, announce=True, *args, **kwargs):
    super(Loop, self).__init__(name, *args, **kwargs)
    self.iterations = kwargs['iterations']
    self.announce = announce
    self.loop_count = 0
    self.name = name
    print("Loop iterations: " + str(self.iterations))

  def run(self):
    while True:
      if self.iterations != -1 and self.loop_count >= self.iterations:
        return TaskStatus.SUCCESS
    for c in self.children:
      while True:
        c.status = c.run()
        if c.status == TaskStatus.SUCCESS:
          break
        return c.status
      c.reset()
    self.loop_count += 1
    if self.announce:
      print(self.name + " COMPLETED " + str(self.loop_count) + " LOOP(S)")

class IgnoreFailure(Task):
  """
    Always return either RUNNING or SUCCESS.
  """
  def __init__(self, name, *args, **kwargs):
    super(IgnoreFailure, self).__init__(name, *args, **kwargs)

  def run(self):
    for c in self.children:
      c.status = c.run()
      if c.status == TaskStatus.FAILURE:
        return TaskStatus.SUCCESS
      else:
        return c.status
    return TaskStatus.SUCCESS

class AutoRemoveSequence(Task):
  """
    Remove each successful subtask from a sequence
  """
  def __init__(self, name, *args, **kwargs):
    super(AutoRemoveSequence, self).__init__(name, *args, **kwargs)

  def run(self):
    for c in self.children:
      c.status = c.run()
      if c.status == TaskStatus.FAILURE:
        return TaskStatus.FAILURE
      if c.statuss == TaskStatus.RUNNING:
        return TaskStatus.RUNNING
      try:
        self.children.remove(self.children[0])
      except:
        return TaskStatus.FAILURE
    return TaskStatus.SUCCESS

class loop(Task):
  """
    Loop over one or more subtasks a given number of iterations
  """
  def __init__(self, task, iterations=-1):
    new_name = task.name + "_loop_" + str(iterations)
    super(loop, self).__init__(new_name)
    self.task = task
    self.iterations = iterations
    self.old_run = task.run
    self.old_reset = task.reset
    self.old_children = task.children
    self.loop_count = 0
    print("Loop iterations: " + str(self.iterations))

  def run(self):
    if self.iterations != -1 and self.loop_count >= self.iterations:
      return TaskStatus.SUCCESS
    print("Loop " + str(self.loop_count))
    while True:
      self.status = self.old_run()
      if self.status == TaskStatus.SUCCESS:
        break
      else:
        return self.status
    self.old_reset()
    self.loop_count += 1
    self.task.run = self.run
    return self.task

class ignore_failure(Task):
  """
    Always return either RUNNING or SUCCESS.
  """
  def __init__(self, task):
    new_name = task.name + "_ignore_failure"
    super(ignore_failure, self).__init__(new_name)
    self.task = task
    self.old_run = task.run

  def run(self):
    while True:
      self.status = self.old_run()
      if self.status == TaskStatus.FAILURE:
        return TaskStatus.SUCCESS
      else:
        return self.status
    self.task.run = self.run
    return self.task

def print_tree(tree, indent=0):
  """
    Print an ASCII representation of the tree
  """
  for c in tree.children:
    print " " * indent, "-->", c.name
    if c.children != []:
```

```
    print_tree(c, indent+1)

def print_phpsyntax_tree(tree):
  """
    Print an output compatible with ironcreek.net/phpSyntaxTree
  """
  for c in tree.children:
    print "[" + string.replace(c.name, "_", "."),
    if c.children != []:
      print_phpsyntax_tree(c),
    print "]",
```

Listing APPENDIX C.5 – Core classes for implementing Behavior Trees in Python [17].

```
#!/usr/bin/env python
import rospy
import actionlib
from actionlib_msgs.msg import GoalStatus
from pi_trees_lib.pi_trees_lib import *

class MonitorTask(Task):
  """
    Turn a ROS subscriber into a Task.
  """
  def __init__(self, name, topic, msg_type, msg_cb, wait_for_message=True, timeout=5):
    super(MonitorTask, self).__init__(name)
    self.topic = topic
    self.msg_type = msg_type
    self.timeout = timeout
    self.msg_cb = msg_cb
    rospy.loginfo("Subscribing to topic " + topic)
    if wait_for_message:
      try:
        rospy.wait_for_message(topic, msg_type, timeout=self.timeout)
        rospy.loginfo("Connected.")
      except:
        rospy.loginfo("Timed out waiting for " + topic)
    # Subscribe to the given topic with the given callback function executed via run()
    rospy.Subscriber(self.topic, self.msg_type, self._msg_cb)

  def _msg_cb(self, msg):
    self.set_status(self.msg_cb(msg))

  def run(self):
    return self.status

  def reset(self):
    pass

class ServiceTask(Task):
  """
    Turn a ROS service into a Task.
  """
```

```python
  def __init__(self, name, service, service_type, request, result_cb=None, wait_for_service=True, timeout=5):
    super(ServiceTask, self).__init__(name)
    self.result = None
    self.request = request
    self.timeout = timeout
    self.result_cb = result_cb
    rospy.loginfo("Connecting to service " + service)
    if wait_for_service:
      rospy.loginfo("Waiting for service")
      rospy.wait_for_service(service, timeout=self.timeout)
      rospy.loginfo("Connected.")
    # Create a service proxy
    self.service_proxy = rospy.ServiceProxy(service, service_type)

  def run(self):
    try:
      result = self.service_proxy(self.request)
      if self.result_cb is not None:
        self.result_cb(result)
      return TaskStatus.SUCCESS
    except:
      rospy.logerr(sys.exc_info())
      return TaskStatus.FAILURE

  def reset(self):
    pass

class SimpleActionTask(Task):
  """
    Turn a ROS action into a Task.
  """
  def __init__(self, name, action, action_type, goal, rate=5, connect_timeout=10, result_timeout=30, reset_after=False, active_cb=None, done_cb=None, feedback_cb=None):
    super(SimpleActionTask, self).__init__(name)
    self.action = action
    self.goal = goal
    self.tick = 1.0 / rate
    self.rate = rospy.Rate(rate)
    self.result = None
    self.connect_timeout = connect_timeout
    self.result_timeout = result_timeout
    self.reset_after = reset_after
    if done_cb == None:
      done_cb = self.default_done_cb
    self.done_cb = done_cb
    if active_cb == None:
      active_cb = self.default_active_cb
    self.active_cb = active_cb
    if feedback_cb == None:
      feedback_cb = self.default_feedback_cb
    self.feedback_cb = feedback_cb
    self.action_started = False
    self.action_finished = False
    self.goal_status_reported = False
    self.time_so_far = 0.0

    # Goal state return values
    self.goal_states = ['PENDING', 'ACTIVE', 'PREEMPTED',
              'SUCCEEDED', 'ABORTED', 'REJECTED',
              'PREEMPTING', 'RECALLING', 'RECALLED',
              'LOST']
    rospy.loginfo("Connecting to action " + action)
    # Subscribe to the base action server
    self.action_client = actionlib.SimpleActionClient(action, action_type)
    rospy.loginfo("Waiting for move_base action server...")
    # Wait up to timeout seconds for the action server to become available
    try:
      self.action_client.wait_for_server(rospy.Duration(self.connect_timeout))
    except:
      rospy.loginfo("Timed out connecting to the action server " + action)
    rospy.loginfo("Connected to action server")

  def run(self):
    # Send the goal
    if not self.action_started:
      rospy.loginfo("Sending " + str(self.name) + " goal to action server...")
      self.action_client.send_goal(self.goal, done_cb=self.done_cb, active_cb=self.active_cb
          , feedback_cb=self.feedback_cb)
      self.action_started = True

    ''' We cannot use the wait_for_result() method here as it will block the entire
      tree so we break it down in time slices of duration 1 / rate.
    '''
    if not self.action_finished:
      self.time_so_far += self.tick
      self.rate.sleep()
      if self.time_so_far > self.result_timeout:
        self.action_client.cancel_goal()
        rospy.loginfo("Timed out achieving goal")
        return TaskStatus.FAILURE
      else:
        return TaskStatus.RUNNING
    else:
      # Check the final goal status returned by default_done_cb
      if self.goal_status == GoalStatus.SUCCEEDED:
          self.action_finished = True
          if self.reset_after:
            self.reset()
          return TaskStatus.SUCCESS
      elif self.goal_status == GoalStatus.ABORTED:
        self.action_started = False
        self.action_finished = False
        return TaskStatus.FAILURE
      else:
        self.action_started = False
        self.action_finished = False
        self.goal_status_reported = False
        return TaskStatus.RUNNING

  def default_done_cb(self, status, result):
    # Check the final status
    self.goal_status = status
    self.action_finished = True
    if not self.goal_status_reported:
      rospy.loginfo(str(self.name) + " ended with status " + str(self.goal_states[status]))
      self.goal_status_reported = True

  def default_active_cb(self):
    pass

  def default_feedback_cb(self, msg):
    pass

  def reset(self):
    self.action_started = False
    self.action_finished = False
    self.goal_status_reported = False
    self.time_so_far = 0.0
```

Listing APPENDIX C.6 – ROS wrappers for behavior trees python library [17].

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist
from rbx2_msgs.srv import *
from pi_trees_ros.pi_trees_ros import *
from rbx2_tasks.task_setup import *

class Patrol():
  def __init__(self):
    rospy.init_node("patrol_tree")
    rospy.on_shutdown(self.shutdown)  # Set the shutdown function (stop the robot)
    setup_task_environment(self)  # Initialize a number of parameters and variables
    MOVE_BASE_TASKS = list()  # Create a list to hold the move_base tasks
    n_waypoints = len(self.waypoints)

    # Create simple action navigation task for each waypoint
    for i in range(n_waypoints + 1):
      goal = MoveBaseGoal()
      goal.target_pose.header.frame_id = 'map'
      goal.target_pose.header.stamp = rospy.Time.now()
      goal.target_pose.pose = self.waypoints[i % n_waypoints]
      move_base_task = SimpleActionTask("MOVE_BASE_TASK_" + str(i), "move_base",
          MoveBaseAction, goal, reset_after=False)
      MOVE_BASE_TASKS.append(move_base_task)
```

```python

    # Set the docking station pose
    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = 'map'
    goal.target_pose.header.stamp = rospy.Time.now()
    goal.target_pose.pose = self.docking_station_pose

    # Assign the docking station pose to a move_base action task
    NAV_DOCK_TASK = SimpleActionTask("NAV_DOC_TASK", "move_base", MoveBaseAction, goal, reset_after=True)

    BEHAVE = Sequence("BEHAVE") # Create the root node
    STAY_HEALTHY = Selector("STAY_HEALTHY") # Create the "stay healthy" selector
    LOOP_PATROL = Loop("LOOP_PATROL", iterations=self.n_patrols) # Create the patrol loop decorator

    # Add the two subtrees to the root node in order of priority
    BEHAVE.add_child(STAY_HEALTHY)
    BEHAVE.add_child(LOOP_PATROL)

    PATROL = Iterator("PATROL") # Create the patrol iterator

    # Add the move_base tasks to the patrol task
    for task in MOVE_BASE_TASKS:
      PATROL.add_child(task)

    LOOP_PATROL.add_child(PATROL) # Add the patrol to the loop decorator

    # Add the battery check and recharge tasks to the "stay healthy" task
    with STAY_HEALTHY:
      # The check battery condition (uses MonitorTask)
      CHECK_BATTERY = MonitorTask("CHECK_BATTERY", "battery_level", Float32, self.check_battery)
      # The charge robot task (uses ServiceTask)
      CHARGE_ROBOT = ServiceTask("CHARGE_ROBOT", "battery_simulator/set_battery_level", SetBatteryLevel, 100, result_cb=self.recharge_cb)
      # Build the recharge sequence using inline construction
      RECHARGE = Sequence("RECHARGE", [NAV_DOCK_TASK, CHARGE_ROBOT])
      # Add the check battery and recharge tasks to the stay healthy selector
      STAY_HEALTHY.add_child(CHECK_BATTERY)
      STAY_HEALTHY.add_child(RECHARGE)

    # Display the tree before beginning execution
    print "Patrol Behavior Tree"
    print_tree(BEHAVE)

    # Run the tree
    while not rospy.is_shutdown():
      BEHAVE.run()
      rospy.sleep(0.1)

  def check_battery(self, msg):
    if msg.data is None:
      return TaskStatus.RUNNING
    else:
      if msg.data < self.low_battery_threshold:
        rospy.loginfo("LOW BATTERY - level: " + str(int(msg.data)))
        return TaskStatus.FAILURE
      else:
        return TaskStatus.SUCCESS

  def recharge_cb(self, result):
    rospy.loginfo("BATTERY CHARGED!")

  def shutdown(self):
    rospy.loginfo("Stopping the robot...")
    self.move_base.cancel_all_goals()
    self.cmd_vel_pub.publish(Twist())
    rospy.sleep(1)

if __name__ == '__main__':
  tree = Patrol()
```

Listing APPENDIX C.7 – The Patrol Robot Scenario Script [17].

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from rbx2_msgs.srv import *
from pi_trees_ros.pi_trees_ros import *
from rbx2_tasks.task_setup import *
from rbx2_tasks.clean_house_tasks_tree import *
from collections import OrderedDict
from math import pi, sqrt
import time
import easygui

# A class to track global variables
class BlackBoard():
  def __init__(self):
    # A list to store rooms and tasks
    self.task_list = list()
    # The robot's current position on the map
    self.robot_position = Point()

black_board = BlackBoard() # Initialize the black board

# Create a task list mapping rooms to tasks.
black_board.task_list = OrderedDict([
  ('living_room', [Vacuum(room="living_room", timer=5)]),
  ('kitchen', [Mop(room="kitchen", timer=7)]),
  ('bathroom', [Scrub(room="bathroom", timer=9), Mop(room="bathroom", timer=5)]),
  ('hallway', [Vacuum(room="hallway", timer=5)])
```

```
 ])

class UpdateTaskList(Task):
  def __init__(self, room, task, *args, **kwargs):
    name = "UPDATE_TASK_LIST_" + room.upper() + "_" + task.name.upper()
    super(UpdateTaskList, self).__init__(name)
    self.name = name
    self.room = room
    self.task = task

  def run(self):
    try:
      black_board.task_list[self.room].remove(self.task)
      if len(black_board.task_list[self.room]) == 0:
        del black_board.task_list[self.room]
    except:
      pass
    return TaskStatus.SUCCESS

class CheckRoomCleaned(Task):
  def __init__(self,room):
    name = "CHECK_ROOM_CLEANED_" + room.upper()
    super(CheckRoomCleaned, self).__init__(name)
    self.name = name
    self.room = room

  def run(self):
    try:
      if len(black_board.task_list[self.room]) != 0:
        return TaskStatus.FAILURE
    except:
      return TaskStatus.SUCCESS

class CheckLocation(Task):
  def __init__(self, room, room_locations, *args, **kwargs):
    name = "CHECK_LOCATION_" + room.upper()
    super(CheckLocation, self).__init__(name)
    self.name = name
    self.room = room
    self.room_locations = room_locations

  def run(self):
    wp = self.room_locations[self.room].position
    cp = black_board.robot_position
    distance = sqrt((wp.x - cp.x) * (wp.x - cp.x) +
            (wp.y - cp.y) * (wp.y - cp.y) +
            (wp.z - cp.z) * (wp.z - cp.z))
    if distance < 0.15:
      status = TaskStatus.SUCCESS
    else:
      status = TaskStatus.FAILURE
    return status
```

```

class CleanHouse():
  def __init__(self):
    rospy.init_node('clean_house', anonymous=False)
    rospy.on_shutdown(self.shutdown) # Make sure we stop the robot when shutting down
    setup_task_environment(self) # Initialize a number of parameters and variables
    MOVE_BASE = {} # Create a dictionary to hold navigation tasks for getting to each room

    # Create a navigation task for each room
    for room in self.room_locations.iterkeys():
      goal = MoveBaseGoal()
      goal.target_pose.header.frame_id = 'map'
      goal.target_pose.header.stamp = rospy.Time.now()
      goal.target_pose.pose = self.room_locations[room]
      MOVE_BASE[room] = SimpleActionTask("MOVE_BASE_" + str(room.upper()), "move_base",
          MoveBaseAction, goal, reset_after=True, feedback_cb=self.update_robot_position)

    # Create the docking station nav task
    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = 'map'
    goal.target_pose.header.stamp = rospy.Time.now()
    goal.target_pose.pose = self.docking_station_pose
    MOVE_BASE['dock'] = SimpleActionTask("MOVE_BASE_DOCK", "move_base", MoveBaseAction, goal
        , reset_after=True, feedback_cb=self.update_robot_position)

    BEHAVE = Sequence("BEHAVE") # The root node
    STAY_HEALTHY = Selector("STAY_HEALTHY") # The "stay healthy" selector
    CLEAN_HOUSE = Sequence("CLEAN_HOUSE") # The "clean house" sequence

    with STAY_HEALTHY:
      # Add the check battery condition (uses MonitorTask)
      CHECK_BATTERY = MonitorTask("CHECK_BATTERY", "battery_level", Float32, self.
          check_battery)
      # Add the recharge task (uses ServiceTask)
      CHARGE_ROBOT = ServiceTask("CHARGE_ROBOT", "battery_simulator/set_battery_level",
          SetBatteryLevel, 100, result_cb=self.recharge_cb)
      # Build the recharge sequence using inline syntax
      RECHARGE = Sequence("RECHARGE", [MOVE_BASE['dock'], CHARGE_ROBOT])
      # Add the check battery and recharge tasks to the stay healthy selector
      STAY_HEALTHY.add_child(CHECK_BATTERY)
      STAY_HEALTHY.add_child(RECHARGE)

    # Initialize a few dictionaries to hold the tasks for each room
    CLEANING_ROUTINE = {}
    CLEAN_ROOM = {}
    NAV_ROOM = {}
    CHECK_ROOM_CLEAN = {}
    CHECK_LOCATION = {}
    TASK_LIST = {}
    UPDATE_TASK_LIST = {}

    # Create the clean house sequence
```

```python
    for room in black_board.task_list.keys():
      ROOM = room.upper() # Convert the room name to upper case for consistency
      # Initialize the CLEANING_ROUTINE selector for this room
      CLEANING_ROUTINE[room] = Selector("CLEANING_ROUTINE_" + ROOM)
      # Initialize the CHECK_ROOM_CLEAN condition
      CHECK_ROOM_CLEAN[room] = CheckRoomCleaned(room)
      # Add the CHECK_ROOM_CLEAN condition to the CLEANING_ROUTINE selector
      CLEANING_ROUTINE[room].add_child(CHECK_ROOM_CLEAN[room])
      # Initialize the CLEAN_ROOM sequence for this room
      CLEAN_ROOM[room] = Sequence("CLEAN_" + ROOM)
      # Initialize the NAV_ROOM selector for this room
      NAV_ROOM[room] = Selector("NAV_ROOM_" + ROOM)
      # Initialize the CHECK_LOCATION condition for this room
      CHECK_LOCATION[room] = CheckLocation(room, self.room_locations)
      # Add the CHECK_LOCATION condition to the NAV_ROOM selector
      NAV_ROOM[room].add_child(CHECK_LOCATION[room])
      # Add the MOVE_BASE task for this room to the NAV_ROOM selector
      NAV_ROOM[room].add_child(MOVE_BASE[room])
      # Add the NAV_ROOM selector to the CLEAN_ROOM sequence
      CLEAN_ROOM[room].add_child(NAV_ROOM[room])
      # Initialize the TASK_LIST iterator for this room
      TASK_LIST[room] = Iterator("TASK_LIST_" + ROOM)
      # Add the tasks assigned to this room
      for task in black_board.task_list[room]:
        # Initialize the DO_TASK sequence for this room and task
        DO_TASK = Sequence("DO_TASK_" + ROOM + "_" + task.name)
        # Add a CHECK_LOCATION condition to the DO_TASK sequence
        DO_TASK.add_child(CHECK_LOCATION[room])
        # Add the task itself to the DO_TASK sequence
        DO_TASK.add_child(task)
        # Create an UPDATE_TASK_LIST task for this room and task
        UPDATE_TASK_LIST[room + "_" + task.name] = UpdateTaskList(room, task)
        # Add the UPDATE_TASK_LIST task to the DO_TASK sequence
        DO_TASK.add_child(UPDATE_TASK_LIST[room + "_" + task.name])
        # Add the DO_TASK sequence to the TASK_LIST iterator
        TASK_LIST[room].add_child(DO_TASK)

      # Add the room TASK_LIST iterator to the CLEAN_ROOM sequence
      CLEAN_ROOM[room].add_child(TASK_LIST[room])
      # Add the CLEAN_ROOM sequence to the CLEANING_ROUTINE selector
      CLEANING_ROUTINE[room].add_child(CLEAN_ROOM[room])
      # Add the CLEANING_ROUTINE for this room to the CLEAN_HOUSE sequence
      CLEAN_HOUSE.add_child(CLEANING_ROUTINE[room])

    # Build the full tree from the two subtrees
    BEHAVE.add_child(STAY_HEALTHY)
    BEHAVE.add_child(CLEAN_HOUSE)
    # Display the tree before execution
    print "Behavior Tree\n"
    print_tree(BEHAVE)
    rospy.loginfo("Starting simulated house cleaning test")
    # Run the tree
    while not rospy.is_shutdown():
      BEHAVE.run()
      rospy.sleep(0.1)

  def check_battery(self, msg):
    if msg.data is None:
      return TaskStatus.RUNNING
    else:
      if msg.data < self.low_battery_threshold:
        rospy.loginfo("LOW BATTERY - level: " + str(int(msg.data)))
        return TaskStatus.FAILURE
      else:
        return TaskStatus.SUCCESS

  def recharge_cb(self, result):
    rospy.loginfo("BATTERY CHARGED!")

  def update_robot_position(self, msg):
    black_board.robot_position = msg.base_position.pose.position

  def shutdown(self):
    rospy.loginfo("Stopping the robot...")
    self.move_base.cancel_all_goals()
    self.cmd_vel_pub.publish(Twist())
    rospy.sleep(1)

if __name__ == '__main__':
  try:
    CleanHouse()
  except rospy.ROSInterruptException:
    rospy.loginfo("House clearning test finished.")
```

Listing APPENDIX C.8 – The House Cleaning Robot Scenario [17].

```
#!/usr/bin/env python
import rospy
from pi_trees_lib.pi_trees_lib import *
from geometry_msgs.msg import Twist

class Vacuum(Task):
  def __init__(self, room=None, timer=3, *args):
    name = "VACUUM_" + room.upper()
    super(Vacuum, self).__init__(name)
    self.name = name
    self.room = room
    self.counter = timer
    self.finished = False
    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
    self.cmd_vel_msg = Twist()
    self.cmd_vel_msg.linear.x = 0.05

  def run(self):
    if self.finished:
      return TaskStatus.SUCCESS
    else:
      rospy.loginfo('Vacuuming the floor in the ' + str(self.room))

      while self.counter > 0:
        self.cmd_vel_pub.publish(self.cmd_vel_msg)
        self.cmd_vel_msg.linear.x *= -1
        rospy.loginfo(self.counter)
        self.counter -= 1
        rospy.sleep(1)
        return TaskStatus.RUNNING

      self.finished = True
      self.cmd_vel_pub.publish(Twist())
      message = "Finished vacuuming the " + str(self.room) + "!"
      rospy.loginfo(message)


class Mop(Task):
  def __init__(self, room=None, timer=3, *args):
    name = "MOP_" + room.upper()
    super(Mop, self).__init__(name)
    self.name = name
    self.room = room
    self.counter = timer
    self.finished = False
    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
    self.cmd_vel_msg = Twist()
    self.cmd_vel_msg.linear.x = 0.05
    self.cmd_vel_msg.angular.z = 1.2

  def run(self):
    if self.finished:
      return TaskStatus.SUCCESS
    else:
      rospy.loginfo('Mopping the floor in the ' + str(self.room))

      while self.counter > 0:
        self.cmd_vel_pub.publish(self.cmd_vel_msg)
        self.cmd_vel_msg.linear.x *= -1
        rospy.loginfo(self.counter)
        self.counter -= 1
        rospy.sleep(1)
        return TaskStatus.RUNNING

      self.finished = True
      self.cmd_vel_pub.publish(Twist())
      message = "Done mopping the " + str(self.room) + "!"
      rospy.loginfo(message)

class Scrub(Task):
  def __init__(self, room=None, timer=7, *args):
    name = "SCRUB_" + room.upper()
    super(Scrub, self).__init__(name)
    self.name = name
    self.room = room
    self.finished = False
    self.counter = timer
    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
    self.cmd_vel_msg = Twist()
    self.cmd_vel_msg.linear.x = 0.3
    self.cmd_vel_msg.angular.z = 0.2

  def run(self):
    if self.finished:
      return TaskStatus.SUCCESS
    else:
      rospy.loginfo('Cleaning the tub...')

      while self.counter > 0:
        self.cmd_vel_pub.publish(self.cmd_vel_msg)
        self.cmd_vel_msg.linear.x *= -1
        if self.counter % 2 == 5:
          self.cmd_vel_msg.angular.z *= -1
        rospy.loginfo(self.counter)
        self.counter -= 1
        rospy.sleep(0.2)
        return TaskStatus.RUNNING

      self.finished = True
      self.cmd_vel_pub.publish(Twist())
      message = "The tub is clean!"
      rospy.loginfo(message)
```

Listing APPENDIX C.9 – The House Cleaning Robot Scenario (simulated cleaning tasks) [17].

APPENDIX D – TURTLEBOT AND THE SIMULATORS

TurtleBot is an open source robot kit designed for research and education, which use ROS. It is officially proposed by Willow Garage[1] to program using ROS and its basic specifications are the following:

Size - Weight - Performance		Sensors
Dimension	354 x 354 x 420 mm	Color Camera
Weight	6.3 kg	Depth Camera
Wheels Diameter	76 mm	3x Rate Gyro
Max. Payload	5 kg	3x forward bump
Max. Speed	65 cm/s	3x cliff
Max. Rotational Speed	180°/S	2x wheel drop

Table APPENDIX D.1 – TurtleBot basic specifications [52].

One advantage of developing using the TurtleBot is that its physical model is available in the Gazebo simulator. *Gazebo* is a multi-robot simulator for indoor and outdoor three-dimensional environments. It generates both realistic sensor feedback and physically plausible interactions between objects, allowing the robot developer to make experiments (such as test algorithms and design robots) using realistic scenarios [33]. We use both Gazebo and Turtlebot to test our agents and artifacts.

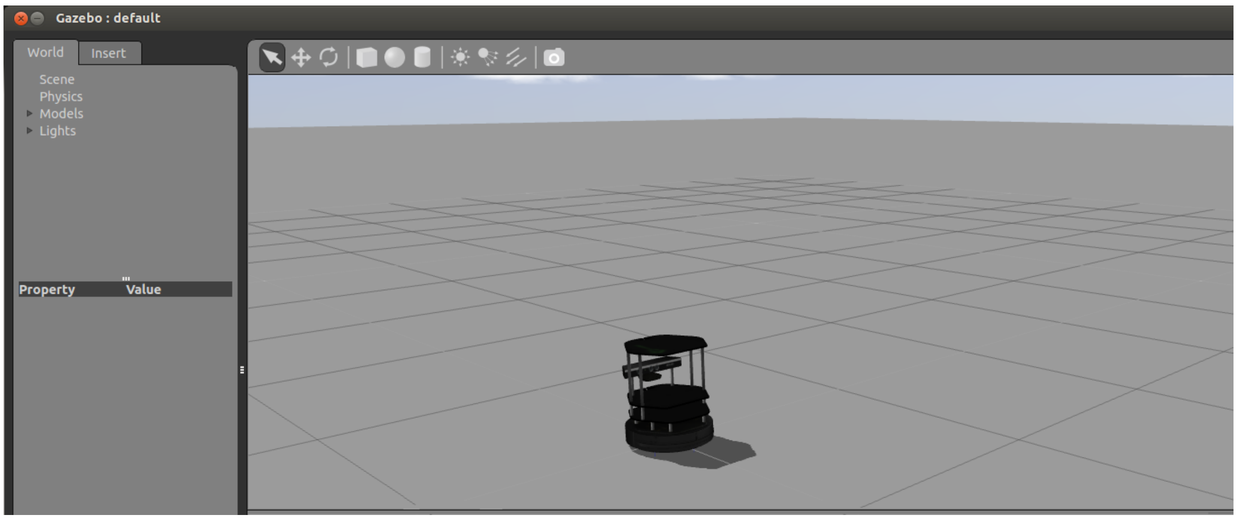

Figure APPENDIX D.1 – Turtlebot simulation on Gazebo

A second simulator used in our project is *Stage*: it provides virtual devices for the robot development frameworks and these simulated devices interact in the same way that real devices do. Stage has two purposes: to enable rapid development of controllers that will eventually drive real robots; and to enable robot experiments without access to the real hardware and environments [15]. Stage needs less hardware requirements to be executed than Gazebo and we notice that this

[1]Willow Garage is a research lab that develops hardware and open source software for personal robotics applications, such as the ROS framework.

characteristic very useful when testing our artifacts for a long time period: Gazebo usually crashes during longer simulations.

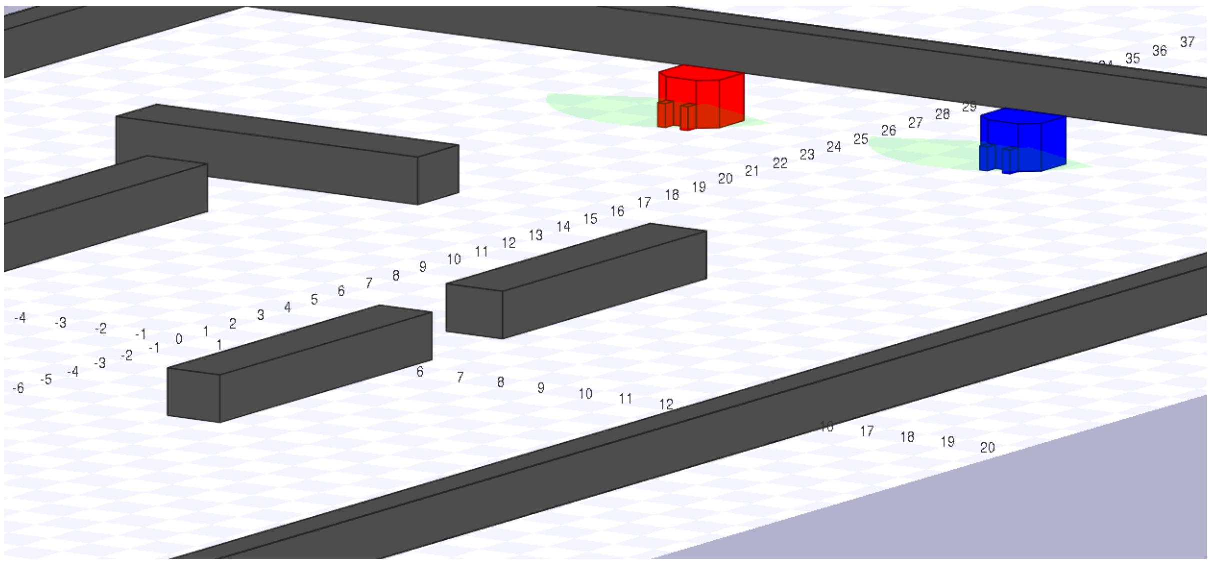

Figure APPENDIX D.2 – Multiple robots on Stage simulator