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

INTEGRATION OF A MULTI-AGENT SYSTEM INTO A ROBOTIC FRAMEWORK: A CASE STUDY OF A COOPERATIVE FAULT DIAGNOSIS APPLICATION

MÁRCIO GODOY MORAIS

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. Alexandre de Morais Amory

Porto Alegre
2015

Dados Internacionais de Catalogação na Publicação (CIP)

M827i Morais, Márcio Godoy
Integration of a multi-agent system into a robotic framework : a case study of a cooperative fault diagnosis application / Márcio Godoy Morais. – Porto Alegre, 2015.
75 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Alexandre de Morais Amory.

1. Informática. 2. Sistemas Multiagentes. 3. Robótica. 4. Tolerância a Falhas (Informática). I. Amory, Alexandre de Morais. II. Título.

CDD 006.39

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

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada *"Integration of a Multi-Agent System Into a Robotic Framework: A Case Study of a Cooperative Fault Diagnostic Application"* apresentada por Márcio Godoy Morais como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 11/03/2015 pela Comissão Examinadora:

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

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

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

Prof. Dr. Jomi Fred Hübner – — UFSC

Prof. Dr. Aurélio Tergolina Salton – — FENG/PUCRS

Homologada em 22/10/2015, conforme Ata No. 019 pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

INTEGRAÇÃO DE UM SISTEMA MULTIAGENTE EM UM FRAMEWORK ROBÓTICO: UM ESTUDO DE CASO DE APLICAÇÃO DE DIAGNÓSTICO DE FALHA COOPERATIVO

RESUMO

A programação de sistemas autônomos multi-robô pode ser extremamente complexa sem o uso de técnicas de desenvolvimento de software apropriadas à abstração de características de hardware, assim como pode ser difícil lidar com a complexidade de software necessária ao comportamento autônomo coordenado. Ambientes reais são dinâmicos e eventos inesperados podem ocorrer, levando um robô a situações não previstas ou até mesmo situações de falha. Este trabalho apresenta um método de integração do sistema multi-agente Jason com o framework robótico ROS. Através desta integração, missões complexas podem ser mais facilmete descritas tendo em vista o uso da linguagem de agentes e seus recursos, bem como a abstração de detalhes de hardware do processo de tomada de decisão. Além disso, módulos de software vinculados ao controle do hardware e módulos com alto consumo de recurso de CPU são separados das rotinas de planejamento e tomada de decisão através de camadas de software, possibilitando o reuso de planos e módulos de software em diferentes missões e robôs. Através desta integração, recursos do sistema multi-agente, tais como a reconsideração de planos e planos de contingência, podem ser utilizados de forma a permitir que o robô reavalie suas ações e estratégias a fim de atingir seus objetivos ou tome ações de forma a lidar com situações imprevistas diante da dinamicidade do ambiente ou quando falhas são detectadas no hardware do robô. A integração permite ainda a cooperação entre múltiplos robôs através de uma linguagem de comunicação padronizada entre agentes. O método proposto é validado através de um estudo de caso aplicado a robôs reais onde um robô pode detectar falhas em seu hardware e diagnosticá-las através da ajuda de outro robô, em um método cooperativo de diagnóstico altamente abstrato.

Palavras Chave: ROS, Jason, diagnóstico de falhas cooperativo, robótica.

INTEGRATION OF A MULTI-AGENT SYSTEM INTO A ROBOTIC FRAMEWORK: A CASE STUDY OF A COOPERATIVE FAULT DIAGNOSIS APPLICATION

ABSTRACT

Programming multi-robot autonomous systems can be extremely complex without appropriate software development techniques to abstract hardware faults, as well as can be hard to deal with the complexity of software required the coordinated autonomous behavior. Real environments are dynamic and unexpected events may occur, leading a robot to unforeseen situations or even fault situations. This work presents a method of integration of Jason multi-agent system into ROS robotic framework. Through this integration, can be easier to describe complex missions by using Jason agent language and its resources, as well as abstracting hardware details from the decision-taken process. Moreover, software modules related to the hardware control and modules which have a high CPU cost are separated from the planning and decision-taken process in software layers, allowing plan and software modules reuse in different missions and robots. Through this integration, Jason resources such as plans reconsideration and contingency plans can be used in a way where they can enable the robot to reconsider its actions and strategies in order to reach its goals or to take actions to deal with unforeseen situations due the environment unpredictability or even some robot hardware fault. The presented integration method also allows the cooperation between multiple robots through a standardized language of communication between agents. The proposed method is validated by a case study applied in real robots where a robot can detect a fault in its hardware and diagnose it through the help of another robot, in a highly abstract method of cooperative diagnosis.

Keywords: ROS, Jason, cooperative fault diagnosis, robotics.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

MTBF. – Mean Time Between Failures

MAS. – Multi-Agent System

SOA. – Service-Oriented Architecture

SLAM. – Simultaneous Localization and Mapping

ROS. – Robot Operating System

SMACH. – ROS State Machine

JADE. – Java Agent Development Framework

FIPA. – Foundation for Intelligent Physical Agent

CONTENTS

1. INTRODUCTION

Advances in technology, availability of software and low-cost hardware that can be applied to robotics have increased the interest in the development of robots around the world. These technological advances allow robots to perform more complex tasks in less controlled environment.

Autonomous mobile robots are being increasingly employed in real-world applications and places such as homes, hospitals, shopping malls and museums. Such robots are designed to perform different tasks and have different types of sensors, actuators, and processing elements, resulting in typically highly heterogeneous hardware platforms. Moreover, the services that the robots provide are also becoming more sophisticated, enabling some degree of autonomy even in unknown or partially known environments.

It is becoming unfeasible to program such sophisticated systems as a single monolithic application. For this reason, several software architectures have been created to enable development of modular software, in which the modules can interact with each other by passing data and messages [11, 20, 17]. This approach facilitates the design of distributed applications, making software complexity manageable and resulting in highly maintainable and reusable software modules.

When robots face the real world, many unpredicted situations may occur. Carlson *et al.* [5] demonstrate that reliability in the field is typically low, between 6 to 24 hours of Mean Time Between Failures - MTBF, even for tele-operated robots. This low reliability requires constant human intervention, which hinders the purpose of using robots in the first place. One common way to increase the reliability is by hardware redundancy. However, it increases the cost and design complexity.

Instead of building a single very expensive robot with extensive use of redundancy to increase reliability, it might be more efficient, economic, and reliable to have multiple simpler robots collaborating in a given task. It is easier and cheaper to find spare parts for simpler robots which reduces the maintenance time and cost. Multiple robots provide parallelism, redundancy, and tolerance to individual robot failure. When multiple robots are performing a single task, this task can be successfully accomplished even if there are some robot failures during the task execution.

Finally, the successful introduction of mobile robots into other applications depends on reducing their design complexity and cost, and increasing their reliability in the field. Thus, these systems must exhibit some degree of fault-tolerance, or the ability to diagnose and recover from the encountered faults.

The motivation of this work is to provide a programming infrastructure which enables hardware abstraction and code reuse through an integration between a multi-agent system and a robotic framework. The agent's plans presented in this work are described in a highly abstract way using an agent programming language, once the hardware aspects are encapsulated in robotic framework, allowing the reuse of plans.

Moreover, by using an agent programming infrastructure along with the robotic framework, it is possible to abstractly describe more elaborated fault detection and diagnosis plans in which, for instance, two or more robots can collaborate to improve fault tolerance.

This work presents a case study of a complex collaborative fault diagnosis behavior in which a faulty robot is not able to find out the source of its fault, i.e. which component is defective. A second robot is called to assist the faulty robot by observing its behavior and giving feedback about the robot's position and pose. Based on the feedbacks provided by the second robot, the faulty robot can find out the defective component.

This work is organized as follows: The next Chapter provides the state of the art in cooperative fault diagnosis in robotics; Chapter 3 presents an overview of fundamental concepts needed to this work; Chapter 4 presents important aspects of the robotic framework and the multi-agent system; Chapter 5 details the proposed integration; Chapter 6 presents a case study of cooperative fault tolerance using real robots; Chapter 7 presents and discusses the results of the case study as well as it lists the challenges of using real robots in the case study; Finally, the last Chapter presents the conclusions, a discussion about the use of this integration and its viability, and it points out some future works.

2. STATE OF THE ART

Mordenti [23] presents a method to programming cognitive robots by integrating JaCa [29] and Webots [22]. The JaCa artifact creates a new TCP/IP socket server on a well-known port and waits for a connection request from the robot platform. When that happens, a new TCP connection is established, the executive artifact can receive raw data from sensors and issue robot commands which are transformed by the functional level in simple mechanical actions. Although JaCa [29] incorporates Jason [3] multi-agent system, the method proposed in [23] only explores tasks related to a single robot/agent, issues as multiple agents and cooperativeness are not addressed by the author.

Rockel *et al.* [27] present a method to integrate Jadex multi-agent system [24] with a multi-robot system Player/Stage [14] by implementing an intermediate layer between them. The intermediate layer embeds data, robot, device and behavior components and manages the transition between a synchronous interface to the robot hardware and an asynchronous interface provided to the MAS. The data component contains central data types used throughout other components. The Robot component contains a generic Robot class from which implemented specialized robots inherit. From the specialized Robot component, Device components are organized according to the robot architecture to interface with Player/Stage client to have mechanical actions properly executed by device drivers. The Behavior component implements a basic set of behaviors for a mobile robot through communication services provided by the MAS and which are used to create information channels to which an agent can subscribe in order to read or publish interesting information to services according to its activities and abilities.

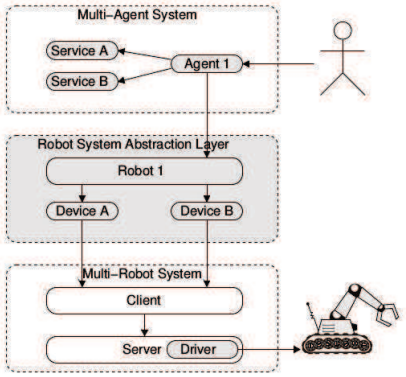


Figure 2.1: Intermediate layer presented by Rockel *et al.* [27].

Boronat [4] proposes ROSAPL framework, an integration of 2APL into ROS. The proposed architecture is divided in two levels where the first level refers to the agent internal architecture and the second level devoted to the whole multi-robot system and its social layer. The agent internal architecture is divided in three layers (Figure 2.2): The Cognitive Layer, implemented using 2APL language; the Operative Layer, based on the environment interface, and the Executive layer, involving hardware and low-to-middle layer control algorithms or advanced robot capabilities provided by ROS packages. Similar to Rockel [27], Boronat [4] implements an intermediate layer

where components such as Action, Capability, and Event are responsible for data exchanging between MAS and middleware. The social level conforms the multi-robot system as a whole. This level is where communication protocols, norms, control mechanisms and other social aspects of the robotic system are defined and implemented.

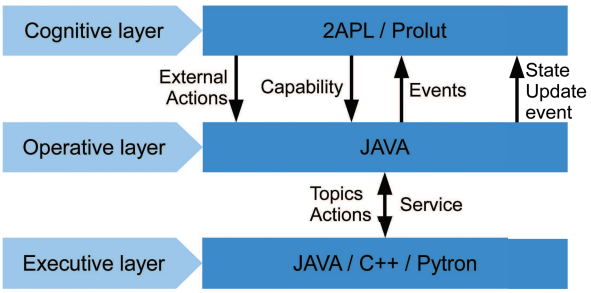


Figure 2.2: ROSAPL Agent programming layers [4].

Crestani and Godary-Dejean [8] present an overview on the issue of fault tolerance in robotics with a focus on the control architecture. Besides analyzing the needs to develop design methodologies for detection, isolation and removal of faults, the article addresses a number of pending issues regarding fault tolerance in robotic systems. Crestani and Godary-Dejean [8] highlight that the methods currently used for fault detection and diagnosis such as signal processing techniques, neural networks, or multivariate statistical techniques are effective but insufficient, since they are not able to address all the issues that should be considered.

It is difficult to ensure that all proposed approaches always satisfy the requirements of real time when a fault occurs. This is an important point for mobile robotics in a dynamic environment where:

- Fault tolerant control routines that integrate diagnosis and fault detection systems and fault recovery mechanisms are efficient while acting on sensors and actuators but are not able to handle faults related to high-level knowledge;
- Despite the efficiency of recovery mechanisms, the proposed solutions are not flexible enough to manage situations experienced during complex missions.

In order to increase the reliability and robustness of fault tolerant robotic systems, Carrasco *et al.* [6] proposes a cooperative method for detection and isolation of faults in homogeneous networks of robots, based on their capabilities and features. This method combines techniques of detection and fault isolation already used in a single robot with ideas presented in cooperative robotic systems. Thus, the local information of a robot is not used solely for its own fault detection and isolation system, but as a redundancy mechanism of devices for any robot in the group. Faults which can be detected by a single robot are determined through probabilistic analysis, and its applicability is limited to faults that fill the requirements of the mathematical model applied, otherwise, another method of isolation and diagnosis of faults is necessary.

In the case of faults which depend on cooperation, they can be checked only if there is proximity between robots and, at least, three valid readings. When two robots are close to each other, sample values of similar sensors are compared and, if the difference between values is above a certain threshold, the robots consider a fault situation. If it is not possible to isolate the fault since they cannot determine which robot has a valid value reading, a third robot is needed to determine which robot actually is in a state of failure. Through the comparison of the analysis of values read from the sensors of the three robots, the fault is isolated considering the disparate value among the three reported ones.

Cooperation among robots is not explored in that paper[17]. Cooperation is limited to the use of the information provided by the robots in form of sensor redundancy.

Currently, the principles of fault tolerance have been neglected in the design of control software and an effort should be made to integrate these principles [8].

P. Iñigo-Blasco *et al.* [17] presented a review of the main aspects of multi-agent robotic systems, which analyzes the common characteristics of robotic frameworks nowadays, their differences and similarities in regard to multi-agent systems.

The authors state that MAS frameworks and middleware used in robotics own tools and offer solutions which are very similar in many aspects, especially those focused on distributed communication architecture. The essential software infrastructure for multi-agent systems is already being implemented by some middleware: support to the development of distributed architectures, methods for exchanging messages between agents, and service-oriented architecture - SOA.

However, most multi-agent robotic systems based communication between agents by developing custom software or general purpose middleware, "reinventing the wheel", instead of using a multi-agent system framework, which is the most appropriate in the development of multi-agent robotic systems. One of the main reasons why the use of multi-agent systems is a good choice for software architecture in robotics is that, when using this approach, the resulting software is more reusable, scalable, and flexible at the same time that the robustness and modularity requirements are kept [17].

There are robotic middlewares that are sufficiently adequate for the implementation of multi-agent robotic systems. This is due to the fact that, despite some shortcomings with regard to the concepts of multiple agents, they provide the infrastructure and tools needed to integrate with multi-agent systems through the creation and deployment of distributed architectures in which they execute components that meet the definition of agent (autonomy, sociability and proactivity).

If the potential of cooperativeness is not fully explored due to a strong relation with hardware characteristics, the integration of robotic framework and multi agent system can offer the necessary flexibility to enable cooperation between robots in a high level, abstracting hardware details. The integration also contributes to the reusability, since it promotes the hardware abstraction, separating hardware details from agents decisions.

3. THEORETICAL GROUNDING

This chapter discusses briefly the key concepts in the development of this work as the *agents, organization of software in robotics, dependability.*

3.1 Agents and Multi-Agent Systems

Wooldridge [31] says there is no universally accepted definition of the term *agent* on account of the importance of different attributes which may vary according to the domain where the term agent is applied. However, the consensus is that autonomy is a central aspect within the concept of *agent*.

Wooldridge [31] postulates the following definition: "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives".

Iñigo-Blasco *et al.* [17] defines an agent, within a multi-agent robotic system, as "autonomous, proactive and social software component", in which the agent can react to external events or messages, and can also be proactive, taking initiative and changing its behavior in order to achieve its goals. This definition meets the list of capabilities that might be expected from an intelligent agent, according to Wooldridge and Jennings [32]:

- *Reactivity* - Intelligent agents are able to perceive their environment, and respond in a timely fashion to changes in order to satisfy their design objectives.
- *Proactiveness* - Intelligent agents are able to exhibit goal-directed behavior by taking the initiative in order to satisfy their design objectives.
- *Social ability* - Intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.

Often the term *autonomy* is used to refer to an agent whose decision making is based mostly on its own perception, instead of the prior knowledge given to it at design time [28]. To achieve some autonomy, the agent must have the ability to perceive itself and the environment around it, and have the means to interact with the environment and other agents.

A key point in the description and understanding of agents was the introduction of the BDI architecture (Beliefs, Desires, Intentions). The basic elements of an agent are some goals the agent has, some information it has, and some deliberation techniques for switching between goals based on this information [10]. The BDI architecture refines this concept and incorporates:

- Beliefs - describing the knowledge of the agent about itself, about other agents and the environment that surrounds it;

- Desires - describing the long-term goals;
- Intentions - describing the goals that have been selected and are being pursued.

Although the concept of agent as an autonomous entity is very important, the paradigm of agent is used in its fullness only when applied to the combined actions of multiple agents[10]. Occasionally the agent faces situations or has complex objectives to which proactivity and knowledge about the environment are not enough. And, in those situations, a group of agents, capable of communicating through a defined communication pattern, may be able to achieve such complex objectives.

Multi-agent system is the system that contains a number of agents, which interact with one another through communication. The agents are able to act in an environment; different agents have different "spheres of influence", in the sense that they will have control over - or at least will be able to influence - different parts of the environment [31].

3.2 Software Organization in Robotics

With the increasing complexity of the applications of robotic systems, the complexity of the software architecture behind such applications increases as well. With this, it is necessary to enhance the software attributes such as modularity, scalability, reusability, efficiency and fault tolerance.

Thus, the main features of a robotic system software are:

- Distributed and concurrent architecture: needed for such architecture to be able to use all the features offered by processors, multiprocessors and microcontrollers, to cover the entire computational cost needed for complex robotic systems;
- Modularity: the system consists of several modules with high cohesion and low coupling to ensure a minimum interdependence and to enable maintainability, scalability and a reusable architecture. This is especially important in robotics due to the lack of standards and a very close relationship with the hardware, which tends to generate non-reusable systems;
- Robustness and fault-tolerance: The malfunction of a component must not impair the functioning of the entire system and, in return, the system must be able to continue its execution as smoothly as possible with the available resources;
- Efficiency and real-time: Many robotic systems have some sort of real time constraint and such constraints are problematic in distributed software architectures. Because of this, the architecture design should take into consideration the use of software, hardware, protocols and communication mechanisms to ensure compliance with such constraints.

Due to a high level of complexity involved in the development of software for robots, a much used approach in development of robotic software is a tiered architecture. The use of

tiered architectures contributes to having a clear separation of responsibilities, low coupling and high cohesion in a system. Such tiered architecture is typically considered through the distribution of the system in three layers: functional, execution and decision (Figure 3.1) [21].

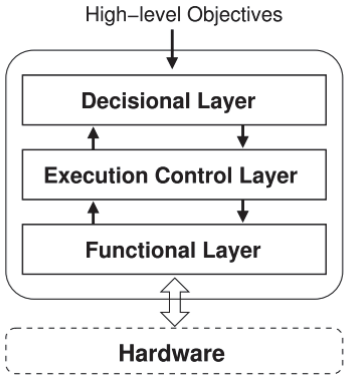


Figure 3.1: Arrangement in a three-tier architecture.

The functional layer is the lowest layer within the three-tier architecture. It is responsible for implementing an interface between the layers of the highest level and the hardware through elementary operations performed on sensors and actuators. In this layer there is no concern with the system or with other modules, but a strong commitment to the extraction of data from sensors and to sending instructions to actuators, to control the hardware efficiently.

The execution layer is responsible for defining a software infrastructure where the exchange of information between layers or modules of the system is performed. Among its responsibilities are the use of raw data aiming to provide resources and services to the system, and the interconnection between the functional layer and the decision layer. An example of resource provided by an execution layer process is the *Simultaneous Localization and Mapping* (SLAM), where the process, based on the sensor's information, constructs or updates a map of an unknown environment while simultaneously keeping track of an agent's location within it, and offers this feature to other system processes.

The decision layer is responsible for the definition of goals, implementation of actions and assessment of either success or failure in the execution of actions, needed to ensure that the objectives are achieved. This layer is responsible for adding reasoning to the robot, and it is dependent on the correct delivery of results by the modules or services provided by the lower layers. An example of resource provided by decision layer process is the use of *ROS State Machine* (SMACH) to control robot behavior.

In order to interconnect the layers of software and enable communication between them through the processes that make up each layer, middlewares[11], which are a software with the potential to manage the heterogeneity of the hardware, were proposed to improve the quality of software application, and to simplify the software design. The developer only needs to implement the logic or algorithm as a middleware component.

A survey presented by Ayssam and Tarek [11] evaluates the main middlewares used in the development of robotic software, their architectures, their objectives, and also evaluates each

one according to a set of attributes such as architecture, simulation environment, standards and technologies, support to distributed and fault detection and recovery. The analyzed middlewares were: Player, CLARAty, ORCA, MIRO, UPNP, RT-Middleware, ASEBA, MARIE, RSCA, OPRoS, ROS, MRDS, OROCOS, SmartSoft, ERSP, Skilligent, Webots, Irobotaware, Pyro, Carmen, and RoboFrame.

Given the context of this work, the ability of fault tolerance and support for distributed environment are considered relevant attributes. The ability of tolerating faults is essential from the moment the robot begins facing the challenges of real life, outside the laboratories and controlled testing environments, and experiencing real critical situations that, in case of failures, can cause damage to the robot or third parties. Support for distributed environment plays a key role as it allows the execution of monitoring processes and the remote extraction and evaluation of data. Furthermore, the use of distributed systems allows easy integration of robots in a network, whether the robots are heterogeneous or not.

Among the analyzed middlewares, the majority does not have any implementation of fault tolerance (ASEBA, MARIE, RSCA, ERSP, Webots, Irobotaware, Carmen, RoboFrame, Pyro) or it is not implemented explicitly (Player, ORCA, MIRO, ROS); A smaller portion is under development (OPRoS, SmartSoft); And a small part implements fault-tolerant system (CLARAty, RT-Middleware, Skilligent). The reasons why we do not use one these fault tolerant-ready frameworks vary according their characteristics: CLARAty has just 10% of its resources available in its latest public release dated from 2007; RT-Middleware is just a standard of robotics platform, its implementations, as OpenRTM-aist, do not develop the fault tolerance component model; and Skilligent is a commercial framework.

Non-explicit fault tolerance refers to features of middleware that offer some type of information that can be used in order to detect an unusual behavior. We can take as an example the Player project, which has a list of exceptions that can be interpreted as faults. ROS, with the application of the concept of modularity, allows the isolation of faults in a single module (*rosnode*), avoiding propagating a fault to the rest of the system. It also has the standard communication channel called *diagnotsic*, used to communicate relevant events or faults of the system [15].

Regarding the ability of distributed processing, almost all frameworks analyzed have this attribute, except Orocos, Pyro, Webots and ERSP.

3.3 Basic Concepts of Dependability

This section presents basic dependability concepts in computing systems, particularly the fault types, the dependability attributes, and the means to reach dependability.

3.3.1 Fault Types

According to Weber *et al.* [30], a system is any entity capable of interacting with other entities, i.e., other systems, whether software, hardware, humans and the physical environment and its natural phenomena. A system failure is an event that occurs when the delivered service deviates from correct service. An error is that part of the system state that can cause a subsequent failure. A fault is the adjudged or hypothesized cause of an error [30, 21].

The fault state can be active, when a particular system or service produces unwanted results, or dormant, when the fault is present, but a key action or situation to make it active has not occurred yet. An example of a dormant fault is to use a memory area which was allocated dynamically without checking the success of the allocation operation. Probably the tests and the execution of the software module will occur within normality until an episode of memory exhaustion occurs, causing an unpredictable behavior of the software module.

Within the context of robotics, faults can be grouped as follows:

- Physical Faults - Physical faults result from natural wear of the hardware components or external factors such as, for example, electromagnetic interference, or extreme temperatures. Some incidents in factories in Japan in the late 80's can be taken as examples: workers died hit or thrown by robots that performed unexpected movements caused by electromagnetic interference[9];

- Design Faults - Design faults can occur during any stage of development, from project conception to completion - acceptance that the software is ready for use. During development, the module under development interacts with the development environment and faults can be introduced through development tools or developers themselves, introducing faults with "malicious" goals or even unintentionally, due to lack of competence [21]. Unintentional design faults are also called "bugs";

- Interaction Faults - Interaction faults are those caused by external elements of the environment interacting with the system. We can consider as interaction fault any external event that, due to the dynamic nature of the environment, prevents the robot from reaching a goal or from interacting with the environment. For instance, an event occurred in the DARPA Urban Challenge autonomous car competition, with autonomous cars from Cornell University and the Massachusetts Institute of Technology[13]. These cars had the execution of their tasks interrupted by inspectors because the Cornell's car was reluctant to decide whether to move or not and the MIT's car, which was close behind, decided to overtake at the exact moment that the Cornell's car made the decision to move. With two cars on a collision course, inspectors were forced to stop both cars under the risk of the cars to collide, causing irreparable damage to both of them.

3.3.2 Dependability Attributes

Dependability of a system is the ability to avoid faults that have a higher severity or that occurs more frequently than acceptable [1] [1]. The term incorporates the following dependability attributes such as reliability, availability, integrity, security, and maintainability [1].

- Reliability: the ability to meet the specification, within defined conditions, for a certain period of operation, and conditioned to be operational at the beginning of the period;
- Availability: a measure of probability that the system is operating in a given instant of time; alternating periods of normal operation and repair;
- Integrity: absence of improper system changes;
- Safety: probability of the system to be either operable and perform its functions, or discontinue their tasks properly so as not to cause harm to its dependents or other systems;
- Maintainability: ability to accept modifications and repairs, i.e., maintenance.

3.3.3 Means to reach dependability

A dependable systems may use to following techniques: fault prevention, fault tolerance, fault removal and fault forecasting. The variations in the emphasis placed on the different attributes directly influence the balance of the techniques to be employed in order to make the resulting system dependable and secure [1].

- Fault forecasting - The goal of fault forecasting is to estimate the current number of occurrences, future incidences and the likely consequences of faults by evaluating the system behavior. The assessment of such behavior occurs through two aspects: Qualitative - It aims to identify and classify fault modes or combinations of events that lead to system failures; and Quantitative - It has the objective of evaluating in terms of probabilities, to the extent that some of the attributes are satisfied. These attributes are then viewed as measures;
- Fault prevention - It deals with preventing failures from occurring or getting introduced into the system. An example of fault prevention method is to improve the development processes so as to reduce the number of faults introduced in the system. In the case of software design faults or software bugs, are used techniques and practices that minimize the insertion of bugs, such as the use of version control, bug tracking, and coding standards;

[1] The dependability specification of a system must include the requirements for the attributes in terms of the acceptable frequency and severity of service failures for specified classes of faults and a given use environment[1].

- Fault removal - Fault removal deals with methods to reduce the number or severity of faults. The removal of faults can occur at two different moments: during the development phase - through the validation process and bug fixing - or during the use of the system - through corrective and preventive maintenance.

- Fault tolerance - A fault tolerant system must be able to detect the occurrence of one or more errors (and possibly faults) and recover itself in order to deliver correct results and isolate such instances of error, preventing the errors from occurring again. For the success of the fault recovery action, the fault tolerance system should be able to manage the occurrence of both, errors and faults.

  - Regarding management of detected errors - error removal can be achieved by saving good system states and performing rollback, rollforward and compensation. Rollback brings the system back to the last saved state (checkpoint) before the occurrence of the error; Rollforward places the system in a reachable and known state, without error; Compensation uses redundant resources or distribution of the workload onto resources without errors, in order to mask the error.
  - With respect to occurrence of faults, four steps are needed: Diagnosis, Isolation, Reconfiguration and Reboot. The diagnosis identifies the type and location of the error that caused the fault. From then, it is required to isolate the source of the fault by the exclusion of components or devices. Given that, at least, one module or device has been deleted, it may be necessary to perform the reconfiguration of the system. Reconfiguration means to reassign the task from a faulty component to a redundant one or to distribute the task among other components. Finally, the information about the new setting is registered and the system goes back into operation.

4. RESOURCES

This chapter introduces briefly the aspects and basic concepts of the two main software components in this work, ROS Robotic Framework in Section 4.1, and Jason Multi-Agent System in Section 4.2.

4.1 ROS - Robot Operating System

ROS - Robot Operating System [25], is a open source robotic framework widely supported by the community that performs research in robotics. The difference between ROS and others frameworks is that ROS does not intend to be a complete framework to replace the others. ROS has been created aiming the reusability of code and resources. As a result, it is flexible and able to use resources, such as drivers and simulators from existing frameworks, and to be used in conjunction with other frameworks. It is a differential in relation to the other frameworks once we can user several resources from different frameworks.

ROS is designed to be modular; a robot control system usually comprises many processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. For example, one process controls a laser range-finder, one process controls the wheel actuators, one process performs localization, one process performs path planning. Processes which use the ROS infrastructure are called *rosnodes*. There is a special *rosnode* named ROS Master which provides node registration and lookup to the rest of the system. The Master also provides the Parameter Server, a shared, multi-variate dictionary that is accessible via network. *Rosnodes* use this server to store and retrieve parameters at runtime. It is globally viewable and the system tools can easily inspect the configuration state of the system and modify if necessary.

Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays. ROS message system uses a simple message descriptor in plain text files (.msg files) to describe messages. Listing 4.1 shows an example of ROS message definition.

```
int32     field1
string[]  field2
float64   field3
```

Listing 4.1: Example of ros message definition.

Each field consists of a type and a name. Field types can be:

- a built-in type (int32, float 32, string, etc);

- nested message descriptors such as "geometry_msgs/PoseStamped";
- fixed-length or variable-length arrays such as float64[] or float64[5];
- the special *Header* type, which maps to std_msgs/Header message descriptor.

At compile time, a preprocessor translates the descriptor file into source code, according to the supported languages used in the package.

Messages are routed via a transport system through a publish / subscribe model. A node sends out a message by publishing it to a given topic. Topics are named, unidirectional and strongly typed communication channels. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption.

Subscriptions and advertisements are managed by ROS Master. The Master stores all information about publishers, subscribers, topics currently in use as well as the type of the messages sent over the topics. After registering the *rosnode* as a publisher or subscriber to a topic in *rosmaster*, all message exchanging is peer-to-peer, messages are sent directly from a publisher to the subscribers. Once a *rosnode* publishes the first message on a *rostopic*, defining its type, any other *rosnode* that tries to publish a different message type receives an error message and the message is dropped. The only valid message type for a *rostopic* is known by *rosmaster*. The *rosmaster* also provides name registration and lookup to the rest of the Computation Graph. Without the Master, *rosnodes* would not be able to find each other, exchange messages or invoke services.

The publish / subscribe model is a very flexible communication paradigm but this model many-to-many one-way transport is not appropriate for request / reply interactions, often required in a distributed system. Request / reply is done through *rosservice*. A *rosnode* offers a service and a client *rosnode* calls the service by sending the request message awaiting the reply. The request and reply messages are defined in a plain text descriptor file (.srv file) similar to messages. Listing 4.2 shows an example of ROS service definition. The fields before the '—' are related to the request message and the fields after '—' are related to the reply message.

```
int32   request_field1
int32   request_field2
---
int32   reply_field
```

Listing 4.2: Example of ros service definition.

ROS, through its philosophy of code and resource reuse, its concept of nodes and distributed system, and its support to different programming languages, offers the modularity necessary for proper separation of responsibilities of processes and the flexibility necessary for integration of

different resources. In this work, such features are fundamental to the integration of a multi-agent system.

4.2 Jason

Jason [3] is an extension of AgentSpeak [26], agent-oriented programming language based on the BDI architecture and used to program the behavior of individual autonomous agents. Jason extends AgentSpeak by featuring inter-agent communication, support for developing environments, possibility of run a MAS distributed over a network, possibility of user-defined internal actions, among others.

As seen in section 3.1, it is expected that agents are reactive when perceiving their environment. The way how they react to events is by executing plans. Plans are described in three parts [3] :

- event - Represents changes, either in beliefs or goals;
- context - Defines when a plan should be considered applicable;
- body - When the event matches the plan's event and the context of the plan is true, the content of body is executed in sequence. The body comprises actions, Achievement goals, Test Goals, Internal actions, and others.

Multi-agent systems are focused on dynamic and often unpredictable environment. In such environments, it is common for a plan to fail to achieve the goal. Therefore, in the same way agents have plans to achieve a goal, they could have fault plans for what to do when a preferred plan fails. While events for achievement goal plans are preceded by "+!", fault plans, also known as contingency plans, are preceded by "-!". Thus, the syntax for agents plans is +!*event* : *context* <- *body* for achievement goal plans and -!*event* : *context* <- *body* for contingency plans.

The agent architecture has an interpreter that runs an agent program by means of a reasoning cycle which is divided in several steps. During its reasoning cycle, an agent perceives the environment, updates the belief base, communicates with other agents, and executes intentions, among other tasks.

In order to integrate robotic framework and multi-agent system, it is necessary to have means to extract actions from the current intention and to send them to the processes in charge of executing the tasks related to the action, as well as means to create new perceptions to Jason agents. Through its extendable agent architecture class, Jason exposes the methods needed to make the integration viable. The class method *act* allows sending messages to the other processes as well as putting the agent and its current intention in a coherent state until it receives an action feedback. The class method *perceive* allows the messages from other nodes to be translated into

agent perceptions, as well as giving some feedback from the current action to the agent and, if necessary, triggering some events.

Jason also allows the use of customized infrastructure. It plays a key role since a multi-agent robotic system is wanted and, through an infrastructure based on JADE [2], it is possible to run Jason in a distributed way, allowing every pair of agent-robot to send messages to each other.

JADE [2] is a software framework to develop agent applications in compliance with the FIPA [12] specifications for interoperable intelligent multi-agent systems. The goal is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents. It 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.

Another aspect of Jason that deserves mention in the integration with middleware is that it requires no modifications in the source code to use the custom class of agent architecture or system infrastructure once Jason allows the customization of most aspects of an agent or a multi-agent system through its project file. Listing 4.3) shows a project file using distributed system infrastructure through *Jade* instead of using *Centralised* infrastructure and declares two agents *turle1* and *turtle2*, both with custom agent architecture *org.ros.jason.RosAgArch* presented in Section 5.2.

```
MAS robots {

    Jade(main_container_host("192.168.0.121"))

    agents:

        turtle1 agentArchClass org.ros.jason.RosAgArch at "c1";
        turtle2 agentArchClass org.ros.jason.RosAgArch at "c2";
}
```

Listing 4.3: Jason project file - Distributed communication and custom agent architecture.

Thus, characteristics as extendable agent architecture, possibility of using different infrastructures, ease of configuration through a project file, and the fact of Jason is a widespread MAS, make Jason the best choice to be used in this work.

5. SYSTEM ARCHITECTURE

This chapter presents Rason, a standardized interface to integrate ROS robotic framework and Jason multi-agent system. The interface allows Jason agents to connect to ROS framework and communicate with other *rosnodes* in the system. The proposed standards define the rules to exchange information between *rosnodes* and Jason agents. These rules comprise topics, messages and content expected by Rason to connect Jason agents and all the robotic system. The overall system architecture is illustrated in Figure 5.1 and detailed afterwards.

The rest of this chapter is organized as follows: section 5.1 describes topics and messages used to communicate Jason agents and ROS nodes; section 5.2 describes the implementation in Java that allows Jason to be a *rosnode*; section 5.3 lists the rules a decomposer *rosnode* must follow to execute its tasks properly; section 5.4 lists the rules a synthesizer *rosnode* must follow to generate valid perceptions to Jason agents.

A robot can have one or more agents running in MAS and it is possible to have standard agents and custom agents running concurrently. Each agent might have its own sets of plans and goals. For instance, it is possible to have an agent dealing with the robot's navigation and a second agent controlling a gripper. Agents using Rason agent architecture are identified in ROS by the prefix */jason/* followed by the agent name.

Agents can have goals which are completely different from each other, thus, plans to reach those goals have distinct needs, which requires access to different sensors, actuators, synthesizers and decomposers. Synthesizer nodes are used to process raw data from one or more sensors, generating useful perceptions. Usually, a synthesizer node is a specialized node, dealing with specific data types, generating a predefined set of perceptions. Depending on the perceptions required on the agents' plan, an agent might need multiple synthesizers, or to reuse synthesizers which generate commonly used perceptions. Agents can receive perceptions from several synthesizers for instance, a specialized synthesizer node that generates perceptions about the presence of a robot. By receiving data from different sources, for example, image frame and depth information, this node is able to generate perceptions about the presence of a robot and its pose. A second synthesizer can generate perceptions about navigation and the robot's position in a map, by receiving information from a different set of sensors or processing nodes.

According to Bordini *et al.* [3], plans are courses of actions that agents commit to executing as a consequence of changes. Those high-level actions are executed by sending messages to decomposer nodes. Decomposer nodes are processes which understand one or more agent's high-level actions and translate them into commands or information to other system nodes. For instance, a decomposer node, in charge of moving the robot, knows two high-level actions *move* and *turn*. When the agent wants to perform *turn*, it sends a message to this decomposer node. The decomposer node, which encapsulates the system details, sends all messages needed to perform the turn to the nodes in charge of interfacing with wheel's actuators.

It is important to point out that both synthesizer and decomposer nodes are not limited to communication with nodes from functional layer, those in charge of interfacing the hardware. A high-level action could be translated, for example, in messages to other nodes in the execution layer responsible for the navigation system, the positioning of a robot's arm, or any other action, either simple or complex. Figure 5.1 refers to those *rosnodes* as *executive nodes*.

Jason agents, synthesizer, and decomposer nodes are the main elements of the proposed integration. They communicate through standard topics, to be detailed below.

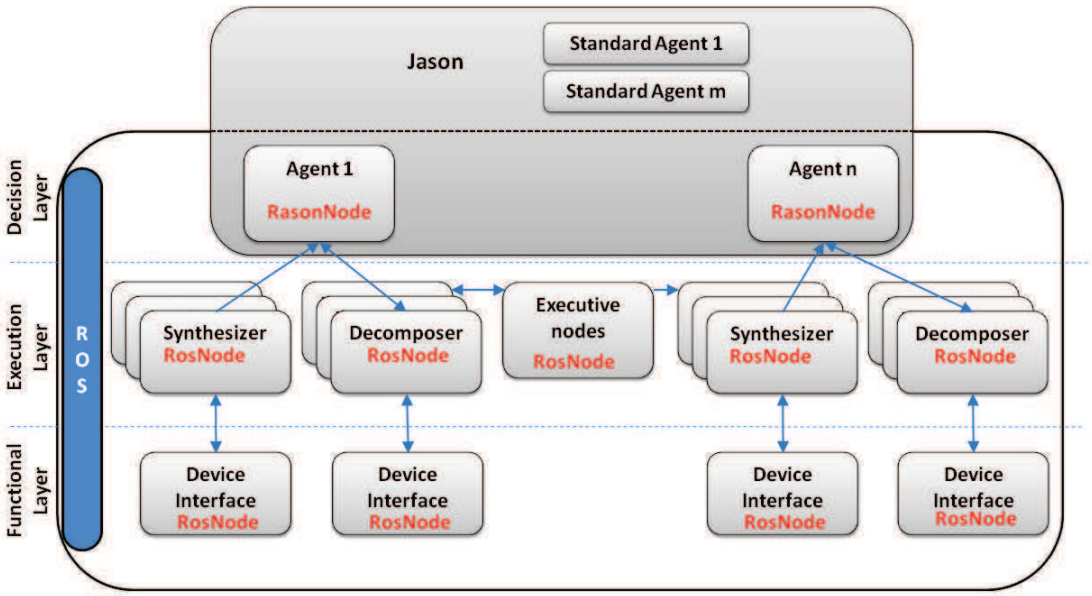


Figure 5.1: Rason, interface to integrate Jason and ROS.

5.1 Agent Interface Topics

The topics and messages described below are the communication channel between Jason agents, synthesizer nodes and decomposer nodes.

The topics have the prefix "/jason/", which identifies topics that provide communication between the ROS nodes and Jason agents, and avoids any kind of collision with topics related to other system activities. Similar to the topics, the messages sent through are grouped in the namespace "jason". Below is the description of each one of the topics as well as its related message.

The topic /jason/action is the communication channel used to send actions from the agent to the decomposer nodes (Figure 5.2). Message jason::action, organized under the definition of ROS message in Listing 5.1, is published on this topic.

```
string agent          # Jason agent which wants to perform the action
int32  action_id      # Internal action identifier
string action         # The action itself
string[] parameters   # Parameters, if any
```

Listing 5.1: action.msg - message definition file for Jason actions.

The fields in listing 5.1 are explained below:

- *agent* - This field identifies the agent which wants to perform an action. It is necessary to give feedback about the action status to the correct agent;
- *action_id* - This field is a unique internal ID, generated when an action message is sent from agent to decomposer nodes. As *agent*, it is needed when a feedback is sent to agent;
- *action* - This field is the action name, i.e., it describes what action should be executed. This field is required by the decomposer to determine if an action must be decomposed or dropped (when another decomposer is responsible for it);
- *parameters* - This field contains the action's parameters. The data in this field are optional, but if present, the decomposer in charge of this action must know how to handle them.

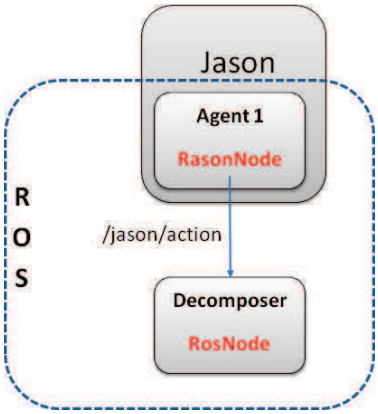


Figure 5.2: Topic to send actions to ROS.

When a *rosnode* needs to notify an agent about internal events, e.g. action completed or occurrence of fault, a message jason::event (Listing 5.2) is sent through the topic /jason/event to the agent (Figure 5.3). This is not a Jason agent event, but an internal event of the proposed integration, which is detailed in Section 5.2.

```
string agent
string event_type
string[] parameters
```

Listing 5.2: event.msg - message definition file for event signaling.

The fields in Listing 5.2 are explained below:

- *agent* - Identifies the receiver agent.
- *event_type* - Specifies the type of the event;
- *parameters* - List of parameters, if any.

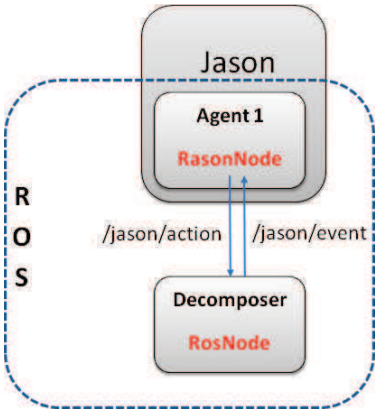


Figure 5.3: Topic to send control events to *RasonNode*.

The topic /jason/perception allows synthesizer *rosnodes* to send useful data, translated into perceptions, to Jason agents. The *rosnode* sends a list of its current perceptions by publishing a jason::perception message (Listing 5.3) on the /jason/perception topic (Figure 5.4). The perceptions sent through the messages in this topic are source of updates in the agent's beliefs.

```
string source          # Source of the perception list
string[] perception    # Perceptions list
```

Listing 5.3: perception.msg - message definition file for Jason perceptions.

The fields in message listing 5.3 are explained below:

- *source* - Identifies the source of the perceptions list. Perceptions are grouped by their sources to avoid overwriting of perceptions sent by other synthesizers in the internal table of the *RasonNode*; This subject is detailed in section 5.2;
- *perception* - List of current perceptions generated by the *rosnode*.

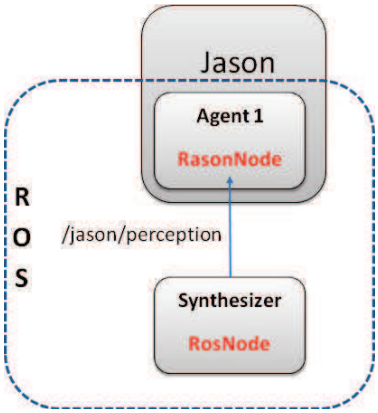


Figure 5.4: Topic to send perceptions to *RasonNode*.

5.2 Rason

The agent architecture is a piece of software which implements a Jason's agent. Since Jason agents need to exchange information with decomposers and synthesizers *rosnodes*, the agent architecture class must be extended to allow the agent to advertise itself as *rosnodes*, retrieve perceptions generated by synthesizers *rosnodes* and send actions to decomposers *rosnodes*. Jason interpreter is developed in Java, thus, the same language for the proposed interface is used.

Developed at Google, in cooperation with Willow Garage, *rosjava* [19] is a pure-java implementation of ROS, aiming at the integration of Android and ROS compatible robots. *rosjava* defines the interfaces and the classes necessary to develop ROS nodes in Java, as well as the tools needed to compile Java source code and to generate message serializers based on ROS *.msg* files. It is necessary the use *rosjava* package since ROS supports officially only C++, Python and Lisp, and the tools provided by ROS core packages are not compatible with Java language.

rosjava defines the interface *AbstractNodeMain*, which must be implemented in order to establish the communication channel with the ROS Master, as well as the other nodes of the framework. Once this interface is implemented, each one of its instances becomes a *rosnode*. The proposed classes and the standards are described below.

RasonNode class implements the *AbstractNodeMain* interface to enable communication through ROS infrastructure, and provides resources to allow the use of a set of agent-related topics and messages by classes in Java.

When a class instance is initialized, it is registered as a *rosnode* on *rosmaster*, advertises itself as a publisher in topic /jason/action, and subscribes to topics /jason/perception and /jason/event, establishing all necessary communication channels to turn a Jason agent into a *rosnode*.

In order to send an action to a decomposer *rosnode*, *RasonNode* exposes *SendAction* method. When invoked, it publishes jason::action messages on the /jason/action topic and returns a unique internal ID associated with the action. The internal ID is important once it gives the agent the ability to identify any event related to the action.

Two function callbacks implement the message listeners for the topics /jason/perception and /jason/event. When called due to the arrival of a new message, the function associated with the topic extracts the message information and stores it in internal tables, to be retrieved by the agent at the appropriate time.

In case of perceptions, the message jason::perception has a field *source* which groups a list of perceptions. Whenever a jason::perception arrives, the data from that source are overwritten in a table of perceptions, keeping only the new ones. This procedure prevents from a burst of messages if a decomposer *rosnode* has a publishing frequency higher than the agent's capability to retrieve those perceptions.

The Jason::event message has a field called *agent* to identify the agent which should receive the message. The message is sent to all subscribers of the topic but only the receiver must process the event. *RasonNode* filters out the messages that are not directed to the associated agent, preventing agents from receiving messages addressed to others.

To retrieve information about internal events and perceptions, the class exposes *GetPerceptions* and *GetEvents* methods, which return lists of the entries stored in the internal tables of the class. Helper functions *ClearPerceptions* and *ClearEvents* are available to clear internal tables of perceptions and events, respectively.

Once a standard for topics and messages has been defined, in section 5.1, and the means of information exchange between Jason agents and ROS nodes are defined, those resources must be available to the Jason agents.

The *AgArch* class implements the default architecture for agents in Jason. It is necessary to extend the agent architecture class in order to give the agent architecture the ability to deal with ROS resources available through *RasonNode* class.

RosAgArch is a specialization of *AgArch* class and it is responsible for the instantiation of *RasonNode* class (each agent has its own private instance of *RasonNode*), for the management of actions currently being performed, and for the perceptions coming from *rosnodes*. The management of actions and perceptions is done through overriding the methods *act*, *reasoningCycleStarting* and *perceive*.

When an agent wants to perform an action, the intention is suspended, the method *act* is called, and a data structure *ActionExec* is passed as a parameter. During its execution, *act* calls the method *SendAction* of *RasonNode* class to send the action to the decomposer, and the *ID* returned by the method is stored in the internal table of tuples *<ID,ActionExec>*. This table is especially important, since *ActionExec* data structure holds information about the action in progress and the suspended intention. This information is required when receiving an event of feedback about the action.

At the beginning of the agent's reasoning cycle, the method *reasoningCycleStarting* is called. *RosAgArch* overrides this method to treat internal events of action feedback coming from *rosnodes*. Those events are sent by decomposer *rosnodes* when an action is finished or an unexpected behavior has occurred. Thus, events whose type is *Action_Feedback* are selected and, after looking

for the *ID* of the action in the internal table, the *ActionExec* data structure is used either to resume the suspended intention, or to trigger a Jason event of fault in the agent's plan. The content of the first occurrence of the *parameters* field indicates the result of the execution of an action. The value *OK* means the action was successful; other values indicate error codes in which different values represent different causes. The second occurrence of the *parameters* field holds the *ID*.

Currently *action_feedback* is the only event mapped in the integration between ROS and Jason. But the structure of internal events has been developed with the intention of supporting other events in the future.

The reason why two *rostopics*, /jason/action and /jason/event, are used to control the actions instead of a single *rosservice* is that an intention is suspended when one of its actions is in progress, but the agent does not stop, it keeps executing its reasoning cycle. According to Bordini *et al.* [3], typically an agent has more than one intention in the set of intentions, each one representing a different focus of attention. These are all competing for the agent's attention, meaning that each one of them can potentially be further executed in the next step of the reasoning cycle. Thus, the use of a *rosservice* and its typical request / replay interactions would block the reasoning cycle.

The third overridden method is *perceive*. This method calls *GetPerceptions* method from *RasonNode* class to retrieve the list of perceptions, updating the agent's belief base. As well as *reasoningCycleStarting*, *perceive* is one of the steps of the reasoning cycle.

5.3 Decomposer nodes

Since Jason and ROS are integrated, it is necessary to describe processes in the execution layer which are able to interpret and to process agent's high-level actions. Thus, decomposer *rosnodes* are responsible for the transition between the agent's high-level actions and the robot architecture.

Robots have different architectures and needs. It makes it impossible to propose a generic decomposer *rosnode*. Those decomposer *rosnodes* must be created based on robot resources and its associated agent's plans. Thus, to perform its duties properly, a decomposer *rosnode* must follow some basic rules:

- When a new jason::action message arrives, the node must check if the *action* field contains an action which it is able to handle, since all subscribers receive messages coming from any publisher.

  Agents can send several different actions by publishing jason::action messages on the topic /jason/action, and several decomposers listen to the same topic. The decomposer can only process messages whose action is known and supported by it. This is checked through the field *action* as shown in Listing 5.1, page 36.

- It must hold both *agent* and *action_id* fields (Listing 5.1, page 36), the decomposer needs to send a feedback to Jason later. Those fields are crucial for *RosAgArch* class instance to resume the suspended intention properly;

- Once all tasks in functional/execution layers are performed, the decomposer needs to provide a feedback to Jason, publishing a jason::event message (Listing 5.2, page 37) on the /jason/event topic and sending back to Jason the fields *agent* and *action_id*. The event must have *action_feedback* assigned to the field *event_type*, and the first occurrence of *parameters* field shall be *OK* in case of successful action or another value in case of fault, when the value represents the reason why the action failed. The second occurrence of *parameters* field holds the *ID* of the current action, received in the *action_id* field of jason::action message.

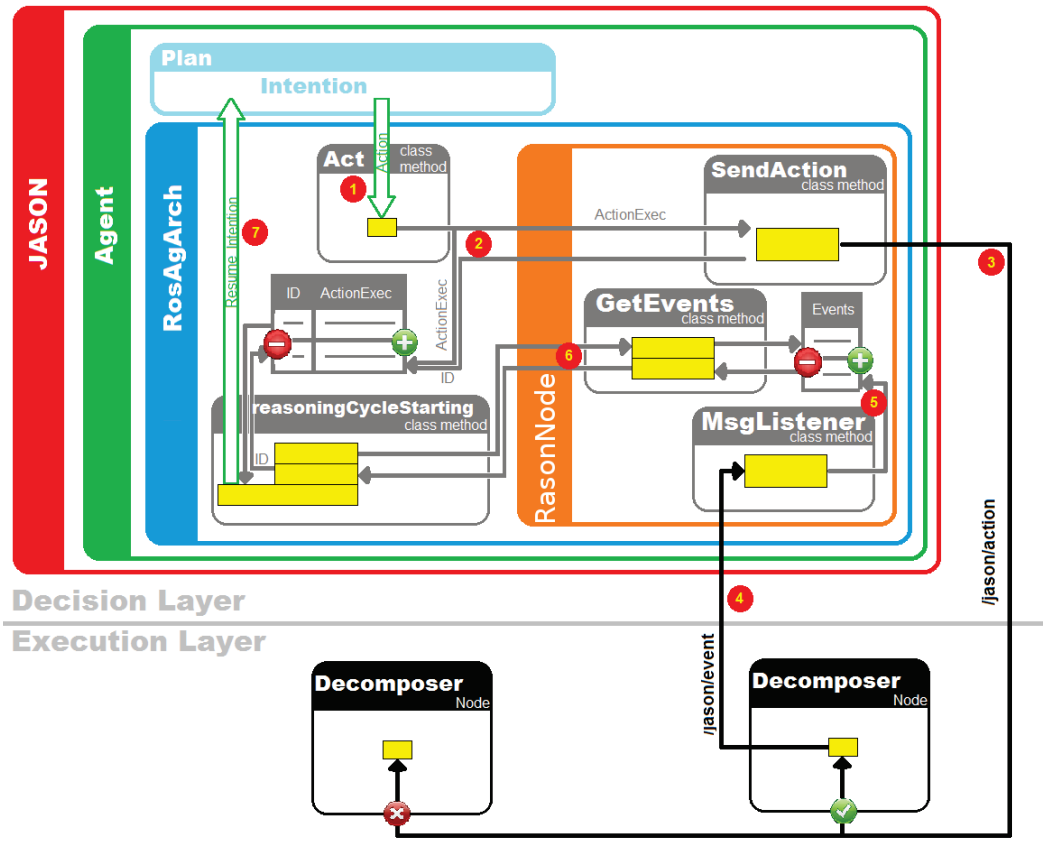


Figure 5.5: Flow of action execution.

Figure 5.5 shows the flow of information exchange between ROS and Jason during the execution of an action. Once the *act* method is invoked due to the execution of an action(1), the method immediately calls the *SendAction* method of its instance of *RasonNode* class. This method returns an internal ID that is stored together with the information of current action (*ActionExec* data structure) in an internal table, in order to retrieve the suspended intention when the action feedback event is received (2) later. Based on the action and its parameters, the jason::action message fields are populated, and the message published on the topic /jason/action(3).

Once the decomposer nodes have received the message, they evaluate whether they are able to process the desired action or not by checking the message field *action*. If not, the message is dropped and the decomposer node remains in its current state, otherwise, the node generates the commands associated with the desired action, monitors the execution of those commands, and publishes a jason::event message with field *event_type action_feedback* on the /jason/event topic(4).

The class method *MessageListener* from *RasonNode* class receives the messages posted in the topic /jason/event and invokes the callback function to treat the message. When the function receives an event message, the event data is stored in an internal table(5) until the agent, through its architecture class instance, retrieves that information through a call to *GetEvents* method(6), performed at the beginning of a new reasoning cycle of the agent, in *reasoningCycleStarting* method. The retrieved information, then, is used to resume the suspended intention, according to the action feedback reason received in the event data(7).

5.4 Synthesizer nodes

As well as decomposer *rosnodes*, synthesizer *rosnodes* cannot be proposed as a generic synthesizer process, since it also depends on the robot's devices and sensors. These processes must be created according to the availability of resource on the robot and the required agent's perceptions.

A synthesizer node must have a clearly and well defined set of perceptions it can generate, based on the topics it knows and the messages it is able to extract data from. Basically, a synthesizer *rosnode* receives messages from other *rosnodes* through the subscribed topics, processes the content of those messages, and produces useful information to the agents. Such information, in form of a perceptions list, is sent by publishing messages jason::perception on the topic /jason/perception.

The basic rules of a synthesizer *rosnode* are:

- Group perceptions by the *source* field, to ensure the correct perception update in *RosNode*;
- A typical *source* identifier is <*node_name*>_<*keyword*>. The *source* field is especially important since several synthesizer *rosnodes* can generate perceptions concurrently in different publishing frequencies. Without the source field, this sort of situation would make the arrival of a new perception message overwrite the current perception in the internal table when trying to keep just the new one. On the other hand, the non-replacement of old perceptions by the new ones might cause a burst of perceptions and compromise the agent's performance, depending on the message's publishing frequency;
- The perception name must follow Jason's syntax for perceptions. These perceptions are directly forwarded to the agent's perception list as a result of *perceive* method from the agent architecture class instance.

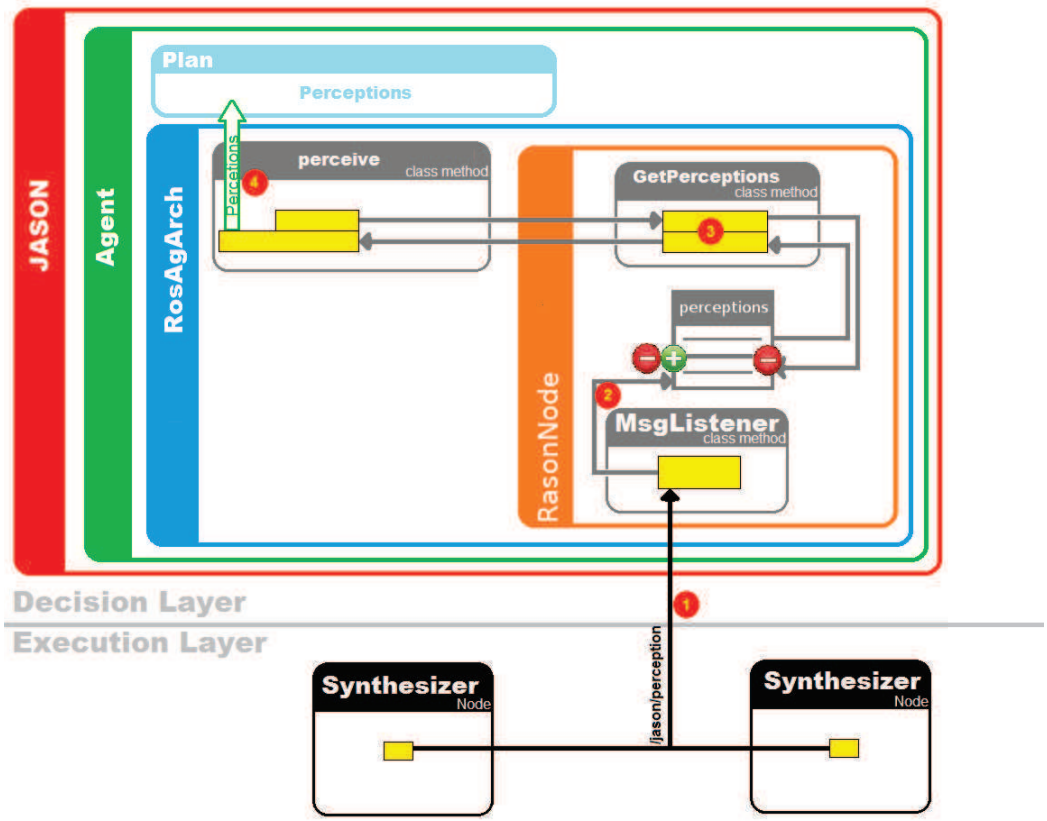


Figure 5.6: Process to create new perceptions.

In the custom architecture of the Jason agent, the instance of *RasonNode* class subscribes to /jason/perception topic and, through the *MessageListener* method, it processes the incoming messages from the synthesizer *rosnodes*(1).

When the *MessageListener* callback method is invoked, perceptions are extracted from the received message and stored in a table of perceptions (2), until the agent requests them by calling the *perceive* method from Jason agent architecture class.

Sensors can generate data at very high frequencies, which might overload the MAS with excessive number of perceptions. Thus, whenever a ROS node, responsible for producing some kind of perception for Jason agent, sends data through jason::perception messages, it also sends the source of these messages. Thus, before storing perceptions in the perceptions table, previous perceptions from that source are removed and only the current ones remain, preventing a burst of perceptions in the agent.

Being the perceptions available on the *RasonNode* of the agent, when the agent requests the perceptions through calls to *perceive* method, this method immediately invokes the method *GetPerceptions* from *RasonNode* class, obtaining the list of new perceptions(3) and allowing the agent to use it in the next steps of the reasoning cycle(4).

5.5 Performance Evaluation

A test plan was developed to evaluate the impact of perceptions bursts in the reasoning cycle time (Listing 5.4). The Jason internal action "time" was modified to return the current time in milliseconds and it was added before and after a loop of a simple action. Based on the elapsed time and the number of loop interactions, it is possible to determine the mean time of the reasoning cycle. The tests were performed by publishing different number of simultaneous perceptions and varying the publishing frequency from those perceptions. Tables 5.1 and 5.2 show the time spent in a reasoning cycle without any perception filtering (Table 5.1) and with perception filtering (Table 5.2) in the *RasonNode*. Images 5.7 and 5.8 show the reasoning cycle time in *RasonNode*. When a filter is applied to incoming perceptions, the time increase tend to be linear (Figure 5.8) and when no sort of filter is applied to incoming perceptions, the time increase tend to be exponential (Figure 5.7).

The tests were performed on a 1.8GHz dual-core AMD64 processor and 4GB of RAM.

```
/* Initial beliefs and rules */
value(0).

/* Initial goals */
!start.

/* Plans */

+!start : .time(X) <- +init(X); !goToDestination.

+!goToDestination : value(X) & X < 3700 <- -+value(X+1); !!
    goToDestination.
+!goToDestination : value(X) & .time(T) & init(I) <- .puts("Done"); .
    print(((T - I)/X)).

+perception1 <- .print("p1").
+perception2 <- .print("p2").
+perception3 <- .print("p3").
+perception4 <- .print("p4").
+perception5 <- .print("p5").
+perception6 <- .print("p6").
+perception7 <- .print("p7").
+perception8 <- .print("p8").
+perception9 <- .print("p9").
+perception10 <- .print("p10").
```

Listing 5.4: Plans to evaluate the impact of perceptions bursts in the Jason's performance.

perceptions \ publising frequency (Hz)	1	3	5	10	30	50
1	0.48	0.50	0.49	0.57	0.89	1.12
3	0.50	0.54	0.56	0.57	1.03	1.24
5	0.54	0.57	0.67	0.83	1.35	1.88
10	0.64	0.99	1.04	1.05	2.26	6.23

Table 5.1: Reasoning cycle time without perception filtering.

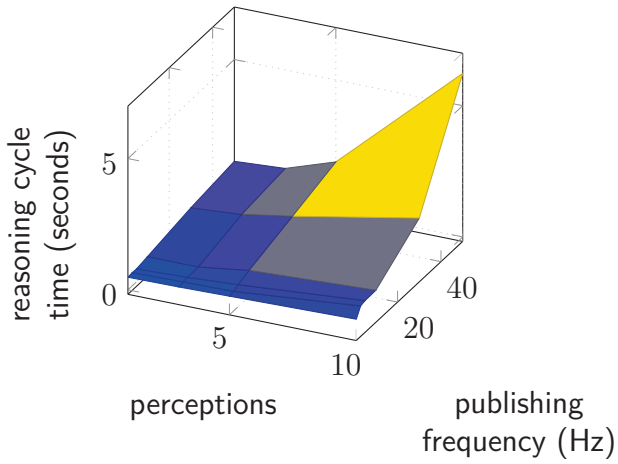


Figure 5.7: Exponential time increase in the reasoning cycle time.

perceptions \ publising frequency (Hz)	1	3	5	10	30	50
1	0.49	0.50	0.53	0.55	0.75	0.85
3	0.52	0.58	0.59	0.71	1.10	1.32
5	0.53	0.60	0.68	0.85	1.25	1.68
10	0.61	0.77	0.85	1.17	2.09	2.58

Table 5.2: Reasoning cycle time with perception filtering.

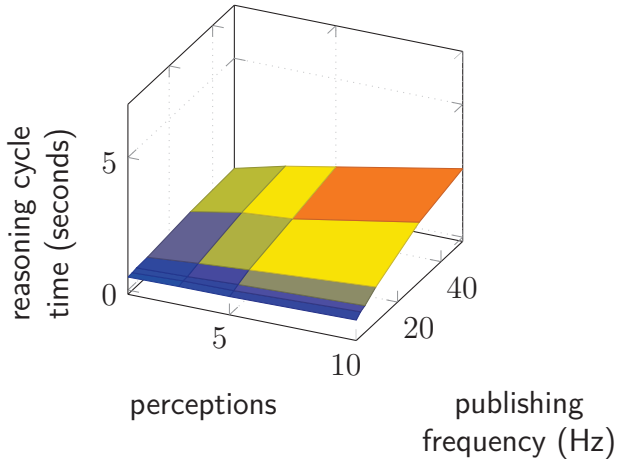


Figure 5.8: Linear time increase in the reasoning cycle time.

Based on results presented in Table 5.1 and Table 5.2, it is possible to note that even in simple tasks like the test plan referred in this section, the number os perceptions as well as their publishing frequencies have direct impact on the system performance. This performance issue reinforces the method proposed in this work, where decomposers and synthesizers nodes in the execution control layer are in charge of the complex algorithms leaving Jason responsible only for the decision-taking process and the task execution control.

Although the proposed method transfers the execution of high computational cost algorithms to the execution control layer, the perceptions generated by synthesizers nodes are sent to every agent in Jason. This perception broadcast can lead the MAS to have a flood of unwanted perceptions, degrading the system performance. We consider as a future work the use of Jason's internal actions to implement a selective method of receiving perceptions, enabling agents to deal only with useful perceptions, improving the system performance.

5.6 Usage Scenarios for RASON

The organization of the main software components within the proposed structure in Rason can be done in two distinct ways: a single instance of Jason and multiples instances of ROS, or multiples instances of both Jason and ROS.

In this work, the system has a single instance of Jason running distributed. One or more agents run within the robot with a local instance of ROS. To enable Jason to run distributed, Jason's project file has been configured to use Jade as MAS infrastructure. Thus, this usage scenario takes advantage of the well developed integration between Jason and Jade which allows Jason to run distributed and also allows robots to communicate each other without any additional programming. Despite being an apparently easy-to-use alternative, some details turns this choice into a difficult option to be used in the field:

- The whole system needs to be configured before using it, i.e., IP address and machine names need to be explicitly declared and associated to robots' computers. This makes impossible the entering of a new robot in the system during the execution;
- When a communication breakdown occurs, the system crashes. This situation can compromise the system reliability;
- There can't be agents with the same name, the reuse of plans in a multi-robot system implies in renaming agents and revising the messages exchanged between agents. Since all agents are visible to each other, there is a risk of an agent mistakenly send a message to the agents of another robot due to human fault during code revision.

Thus, it's necessary the development of another kind of infrastructure, able to fulfill the requirements needed to multi-robot systems when used in the field:

- Each robot has local instances of both, Jason and ROS. The robot is a complete entity which doesn't need any external software elements to manage its interaction with the environment or even the communication with another robots;
- An infrastructure where a robot can freely enter or leave the network without compromised the whole system since it's not uncommon a robot to lose contact when in the field;
- A method to detect the presence as well as the absence of robots in the network, to allow the robot to know whether a certain robot is reachable before sending messages to that robot and to avoid stopping the system execution when a new robot needs to enter the network.;
- The ability to encapsulate and transmit agent messages through different communication channels once different robots can use different hardware to exchange information.

6. CASE STUDY DEVELOPMENT

This chapter presents an application aiming at the validation of the proposed integration by describing a scenario where two robots are navigating in the environment and one of them detects a fault. The robots, then, cooperate to diagnose the fault. The scenario consists of two robots (agents). One is the robot in which the faults are injected, called *Faulty*, and the other is the observer robot, called *Helper*. Each robot is modeled as an agent with its own sets of plans and goals.

Faulty (Figure 6.1(a)) is a simple Turtlebot[1] connected to a notebook[2]. *Faulty* has no devices like camera or sonar attached to itself. *Faulty* has, on its top, two colored cylinders used to identify its pose during the tests. The green cylinder identifies the left side and the orange cylinder identifies the right side.

Helper (Figure 6.1(b)), is also a Turtlebot, but different from *Faulty*, equipped with Kinnect, which enables *Helper* to use computer vision techniques to perceive and help *Faulty*.

(a) *Faulty*. (b) *Helper*.

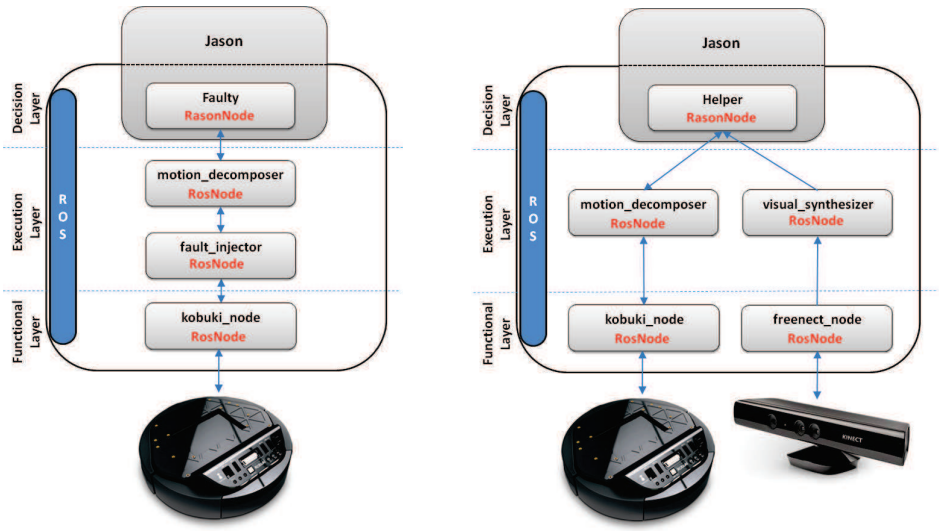
Figure 6.1: Robots used in the experiments

The fault scenarios deal with hardware faults, those related to sensors and actuators and, often, the robot is not able to diagnose them, considering that some faults can cause the same side effects on the behavior of the robot.

The following fault scenarios are planned, based on characteristics of our robots, in order to validate and evaluate the methods of cooperative diagnosis for multiple robots: one of the robot wheels is stuck; both robot wheels are stuck; there is a fault on the wheel encoders.

The ROS nodes are organized as shown in Figure 6.2. Synthesizer and decomposer nodes manage the robot's actions and generate perceptions based on image/depth information available through Kinect. Faults are injected through a service node that intercepts ROS messages to/from Kobuki *rosnode* (turtlebot's ROS-based device driver), in charge of controlling actuators and extracting data from encoders. The details of this figure are described on the following sections.

[1]TurtleBot is a low-cost, personal robot kit with open-source software.
[2]The notebooks are Intel Core I7 with 4Gb of RAM memory



(a) Faulty's nodes. (b) Helper's nodes.

Figure 6.2: ROS nodes organization.

6.1 Hardware

This section presents an overview of the *rosnodes* from functional layer in charge of managing the hardware used in the robots. These *rosnodes* are already available as part of ROS packages.

freenect_node

Microsoft Kinect [16] is a motion sensing input device by Microsoft for Xbox video game consoles and Windows PCs. It enables users to control and interact with their console/computer without the need for controllers, through a user interface using gestures. The device has RGB camera, Depth sensor, microphone and accelerometer.

freenect_node is a functional layer *rosnode* which is a ROS driver for kinect hardware, that acquires data from Kinect through *libfreenect* [7], an open-source library/userspace driver for the Microsoft Kinect, and publishes messages on several topics related with camera calibration, image, depth and combined information such as registered depth camera (aligned with RGB camera) and point clouds. *freenect_node* also offers two *rosservices* to provide calibration for both RGB and IR cameras.

kobuki_node

iClebo Kobuki [33] is a low-cost mobile research base designed for education and research on state of art robotics. Similar to *freenect_node*, *kobuki_node* is a ROS wrapper for the Kobuki driver. *kobuki_node* publishes messages on several topics about hardware diagnosis, events (for instance, a bumper hit) and odometry, it also subscribes to topics related to commands to actuators and its internal odometry system.

6.2 Synthesizer Node

This synthesizer node gives to the agent a perception of the presence of another robot and its coordinates, in order to locate a faulty robot or give it a feedback during a cooperative fault isolation procedure.

The *Helper* has a Microsoft Kinect device which gives the ability to locate and track another robot, giving some feedback of angle, depth and pose in relation to the observer (Figure 6.3).

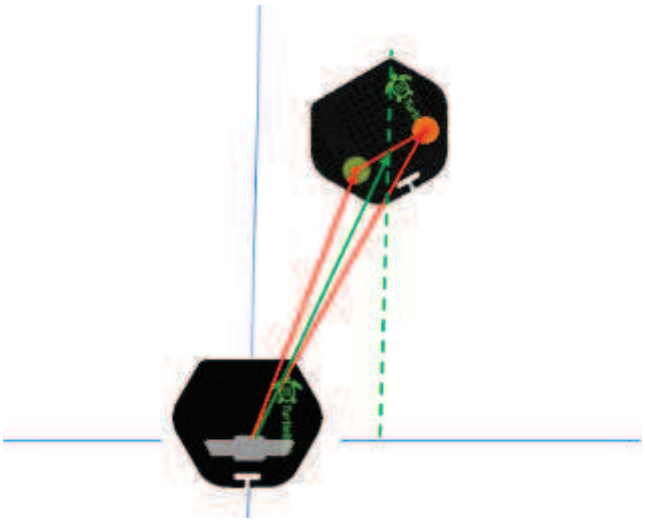


Figure 6.3: A robot observed by other one - angle, depth and pose.

This node subscribes to /camera/rgb/image_raw topic, which holds image data, and /camera/depth_registered/points topic, which holds a depth image point cloud, both topics provide data extracted from a Microsoft Kinect using nodes from a ROS stack that encapsulates *libfreenect*.

When the visual perception node receives both image and depth frame, it tries to recognize the colored cylinders on top of the other robot through computer vision techniques described in Section 6.2.1.

Having identified the regions of interest - the colored cylinders - the node is able to extract information in order to synthesize it and send it as a useful perception to Jason.

The angle of the observed robot is related to the position of the observer (Figure 6.4).

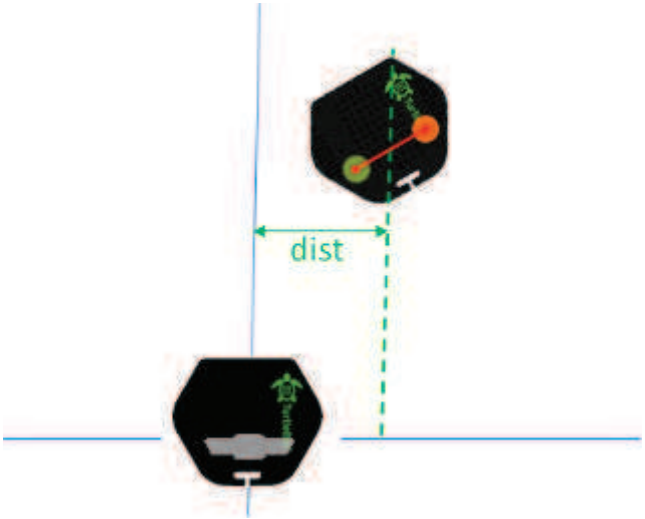


Figure 6.4: Robot's angle estimation.

Once it is known that Kinect has a horizontal field of view of 57°, it is possible to map angular offsets up to 28.5°left or right, from the image center to middle point of the pair of colored cylinders. Thus, *angle* is given as follow:

$$angle = \frac{qrc\_center\_w - (\frac{img\_width}{2})}{\frac{img\_width}{2}} \times 28.5$$

where:

qrc_center_w is the middle-point of a pair of colored cylinders.

img_width is the total number of columns of the image frame.

A negative value means the observed robot is positioned on the left side of the observer, a positive value means the opposite, the observed robot is positioned on the right side of the observer.

Once the colored cylinders have been identified, the centroid of their areas is calculated and their depths extracted from the point cloud (Figure 6.5); the distance *depth* between the robots is the mean value from the depth of the centroid of the region of each cylinder and it is extracted directly from the point cloud.

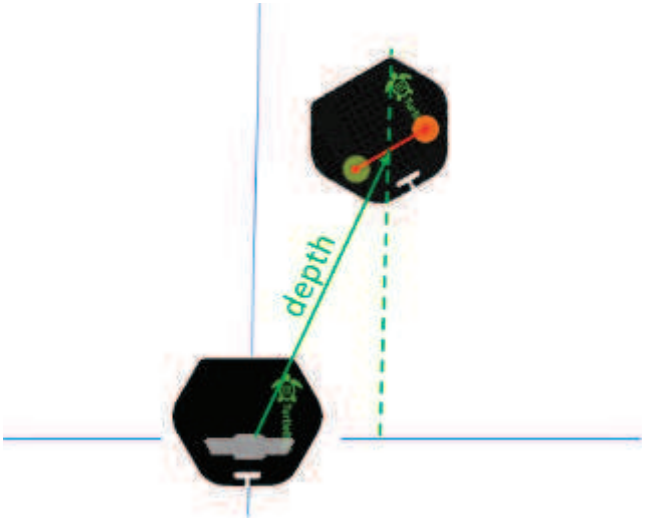


Figure 6.5: Robot's depth calculation.

$$depth = \frac{c1 + c2}{2}$$

Since the distance *H* between the colored cylinders is known (22.7 cm), and the distance from the observer to the centroid of the colored cylinders has been obtained from the point cloud, it is possible to estimate the pose β of the observed robot (Figure 6.6).

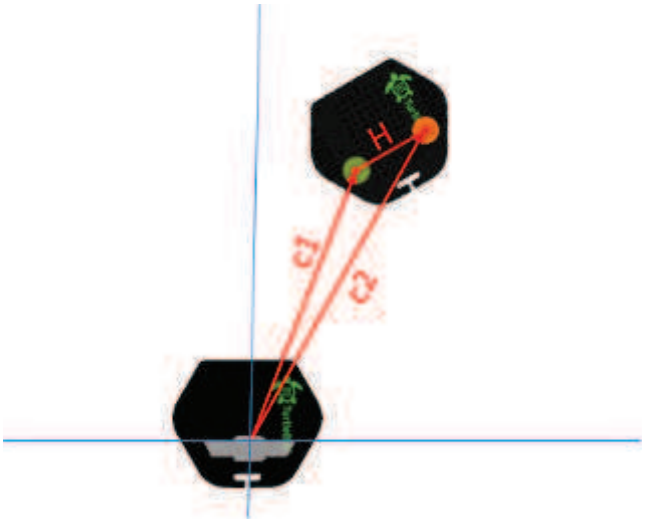


Figure 6.6: Robot's pose estimation.

$$\beta = \arccos(\frac{H}{c2 - c1})$$

Once all necessary parameters have been obtained - Angle, Depth and Pose - the perception, according to Jason syntax, is generated as follows:

robot(*robot_name,angle,depth,yaw*)

After the processes of data extraction and object recognition are finished, a new message is published on the /jason/perception topic, and a list of current perceptions about robots and their attributes - position and distance from the observer - is sent to the agent.

6.2.1 Computer vision routine

The synthesizer *rosnode* subscribes to two topics: /camera/rgb/image_raw to receive image data and /camera/depth_registered/points to receive depth data. The image frame received through the topic /camera/rgb/image_raw is a binary array describing pixels in form of BGR color format, thus, the image frame is converted to HSV color space to allow the use of OpenCV functions. The depth information received through the topic /camera/depth_registered/points is a point cloud, and depth information can be accessed directly.

The colored cylinders are detected by using OpenCV functions to extract blobs[3] according to the desired colors. *InRange* function is used to filter color pixels out by checking if image color pixels lie between lower and upper boundaries. A morphological opening procedure is performed by calling *erode* and *dilate* functions to remove small objects. Next, a morphological closing is performed by calling *dilate* and *erode* to fill small holes. Thus, the remaining shapes in the resulting image are the candidate blobs. This procedure is executed twice, to populate candidate blobs lists to each one of the cylinders.

The procedure to determine the best pair of blobs is executed in two steps. First, the pairs of blobs whose distance is greater than 25 cm or less than 22 cm (22.7 cm is the known distance between cylinders) of distance between them are excluded. Second, among the remaining pairs, the pair whose distance is closer to 22.7 cm is selected.

6.3 Decomposer Node

A motion decomposer node translates the agent's high-level basic motion actions as *move(value)* and *turn(value)* into ROS messages to the nodes of the functional layer which are responsible for interface with the actuators of the robot.

This node subscribes to the topic /jason/actions and handles incoming messages whose high-level action is *move* or *turn*.

In addition to the treatment of the actions themselves, the node monitors the result of commands sent to the functional layer in order to send feedback events back to the decision layer subscribing to /odom topic that publishes the current odometry of the robot.

[3]A blob is a region of a digital image in which some properties are constant or vary within a prescribed range of values; all the points in a blob can be considered in some sense to be similar to each other.

In the case of linear motion, an event of completed action is sent back to the decision layer when the difference between initial and current odometry is greater than or equal to the *value*, being *value* correspondent to a distance in meters.

In the case of circular motion, an event of completed action is sent back to the decision layer when the difference between initial and current odometry is greater than or equal to the *value*, being *value* correspondent to the angle in degrees.

The progress of the movement is monitored by evaluating motion versus time. If there is no change in the odometry by more than a second, a feedback event of failure is sent back to the decision layer with *"no_progress"* as reason for failure.

Another way to monitor the status of the current action is through the trajectory. When the dot product between the vector generated by the start and the end positions, and the vector generated by the initial and the current positions is less than 0.99, a fault event is sent back to the decision layer with the reason *"bad_move"*, which means the robot is taking an unplanned trajectory.

The dot product between two vectors provides a real number which is the result of the product of B by the scalar projection of A in B (Figure 6.7).

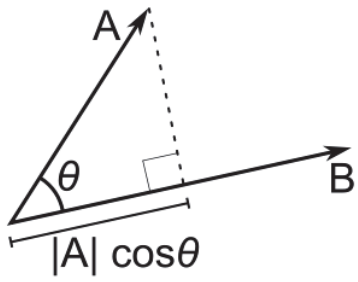


Figure 6.7: Dot product - Projection of vector A in vector B.

Thus, the angle between two vectors is given as follow in Equation 6.1:

$$\theta = arccos\frac{A \cdot B}{|A||B|} \tag{6.1}$$

Equation 6.1: Angle between two vectors using dot product.

The fault injection is done through a ROS Service (messages defined in Listing 6.1) - that intercepts geometry_msgs::Twist messages (see Listing 6.2) published on the topic /cmd_vel and the nav_msgs::odometry messages (see Listing 6.3) published on the topic /odom and, according to the fault type, changes the value of the parameters simulating the behavior of the active fault.

```
string fault            # new fault type or "cancel"
---
string return_message   # return message
```

Listing 6.1: fault_injector.srv - ROS message definition for fault injector service.

This service injects the following faults: fault in one of the actuators, fault in both actuators, fault in the wheel encoder. The fault injection is activated by command line, in which the fault type can be one of the types listed below:

- *left_actuator* - injects fault on the left wheel actuator;
- *right_actuator* - injects fault on the right wheel actuator;
- *both_actuators* - injects fault on both actuators;
- *odometry* - injects encoder fault;
- *cancel* - cancels the current fault injection, if any.

In the case of fault in one of the actuators, the fault injector service, after intercepting the original geometry_msgs::Twist message, alters the value of the angular velocity in Z axis *z_angvel* (Equation 6.2) and decreases the linear velocity in the X axis *x_linvel* (Equation 6.3), making the robot perform a smooth turn to the side of the wheel which has a defective actuator.

$$z\_angvel = \begin{cases} x\_linvel, & \text{if } fault\_type = left\_actuator \text{ and } x\_linvel > 0 \text{ or} \\ & fault\_type = right\_actuator \text{ and } x\_linvel < 0 \\ -x\_linvel, & \text{if } fault\_type = right\_actuator \text{ and } x\_linvel > 0 \\ & fault\_type = left\_actuator \text{ and } x\_linvel < 0 \end{cases} \tag{6.2}$$

Equation 6.2: Angular velocity to simulate an actuator fault.

$$x\_linvel = \frac{x\_linvel}{2} \tag{6.3}$$

Equation 6.3: Linear velocity to simulate an actuator fault.

In case of fault in both actuators, the fault injector service, after intercepting the original geometry_msgs::Twist message, resets the values in the three axes X,Y,Z in both angular and linear movement, making the robot unable to move.

Both linear and angular movements refer to fields of geometry_msgs::Twist message (Listing 6.2) of ROS, responsible for describing the robot's movement. The fields X, Y and Z are fields of complex type *Vector3*.

```
# This expresses velocity in free space
# broken into its linear and angular parts.

Vector3  linear
```

```
Vector3  angular
```

Listing 6.2: Twist.msg - ROS message that describes basic motion action.

When it comes to the encoder fault, the fault injector service intercepts the ROS message nav_msgs::odometry (Listing 6.3) and changes the value of all variables to the last values intercepted before the activation of the fault, causing the subscribers to receive always the same value.

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Listing 6.3: odometry.msg - ROS message that provides odometry information.

6.4 Agents' plans

Faulty has a simple goal: to move forward for about 1 meter and, then to move back for 1 meter, returning to the original position. If the robot cannot complete its goal, a contingency plan is fired to find out the faulty subsystem, enabling a faster maintenance or even self-healing behavior when this is appropriate.

Due to its simple hardware structure (without redundancy) it might not be able to disambiguate the source of fault, since several can be the cause of the same faulty behavior. For instance, a report of no progress coming from the decomposer node can be caused either by a fault in actuators, when the robot remains stopped, or in the encoders, when the robot is moving but the odometry system is unable the compute the distance traveled, always reporting the same value.

On the other hand, a redundant hardware like a giroscope could be used, in this case, to disambiguate between encoder fault and actuator fault, by giving the direction towards which the robot is moving, if any. However, the *Faulty* robot is kept as simple as possible and without such redundant hardware.

If *Faulty* is unable to diagnose the fault by itself, it asks for help by broadcasting a help request to all known agents and awaits some response. *Faulty* and *Helper* are positioned side by side, separated by a distance of 1.5 meters. When *Helper* receives the request, it starts looking for *Faulty*, performing a turn in several steps of 5°.

When *Helper* perceives *Faulty*, it repositions itself and notifies *Faulty* of its availability to help. But, if *Helper* completes the full turn and *Faulty* is not perceived, the process is aborted

by *Helper* and *Faulty* remains awaiting help. From the moment *Faulty* receives the notification from *Helper*, it tries to perform a simple move one meter forward while *Helper* observes that try. Having finished the movement, *Faulty* asks for some feedback from *Helper*, which tells *Faulty* about the initial and current positions. Based on the information received from *Helper* and the data in the notification of fault received from execution layer, *Faulty* tries to identify which component is responsible for the malfunction detected in the system. Thus, it is possible to address three main moments during the scenario: fault detection, robot detection, and diagnosis. Fault detection is not a cooperative step but, as all the other steps, it involves the integration between Jason and ROS, either to move the robot or to generate events of faults to the decision layer. This step begins when *Faulty* starts moving and finishes when *Faulty* asks for help.

The second step initiates the cooperation between the robots. Once *Faulty* has detected a fault and it is not able to diagnose alone, since there is ambiguity according to the nature of the fault, it asks for help, and *Helper* starts looking for *Faulty* through the perception coming from its computer vision synthesizer node.

In the third step, *Faulty* tries to perform the actions needed to disambiguate the diagnosis and, after that, it requests some feedback from *Helper*, that reports position and pose according to its beliefs, based on its own perceptions.

Figure 6.8 shows the message exchange between *Faulty* and *Helper* during the second and third steps. Jason plans are described in next subsections and detailed in Appendix APPENDIX A

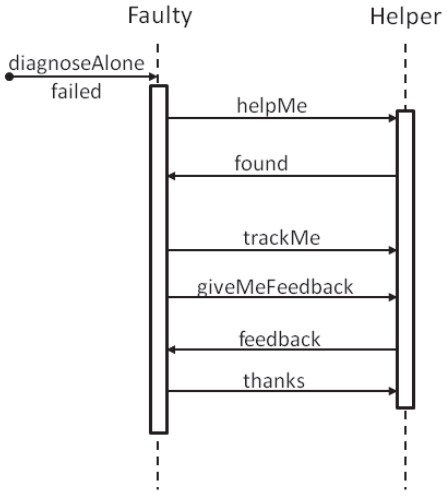


Figure 6.8: Message exchange between *Faulty* and *Helper*.

Faulty's Plans

A Behavior Tree is a model of plan execution used to describe switchings between a finite set of tasks in a modular way. Each node represents a task and a complex task could have several child tasks. There are two key node types in behavior trees: selector and sequencer. The selector, represented by a "?" symbol, executes the next sibling node when the execution of the current node

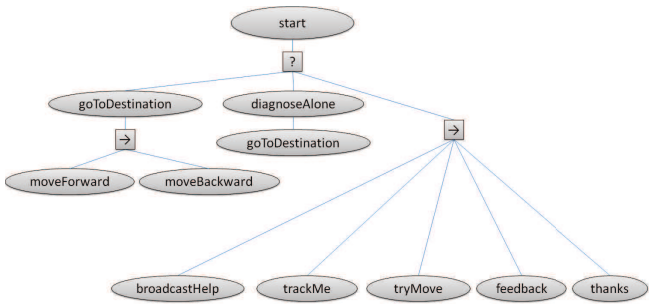


Figure 6.9: Faulty's behavior tree.

fails. The sequencer node, identified by a "→" symbol, executes the next sibling node when the current node has its execution successfully finished.

Figure 6.9 shows a behavior tree that describes the Faulty's plans used to reach the goal as well as plans to deal with its faults. Initially, the faulty robot tries to reach a destination using the goal `goToDestination`. When Faulty detects it is not able to reach the destination (due to the failure of `goToDestination` action), it executes its local fault plan `diagnoseAlone`. The local fault plan for the fault scenarios is empty once Faulty has no devices to disambiguate the source of the fault. When the second attempt to move fails, it starts a cooperative fault detection strategy by broadcasting a help request (`broadcastHelp`). When Faulty receives a positive response from Helper, it asks Helper to track it (`trackMe`) and, when Helper is ready to track it (`tryMove`), Faulty tries to move again. After completing the movement, Faulty asks Helper for feedback on its movement and evaluates the source of the fault(`feedback`). A `thanks` message is sent back to Helper to let it know the help is no more needed.

The resulting movement is calculated according to the fault detected. When Faulty tries to move but no progress is detected, if Helper reports a current position, pose, or depth different than the initial one, it considers the encoders have fault, otherwise either the actuators have faults or the wheels are stuck. When Faulty detects a movement that is in discordance with the expected movement, the composition of Faulty's pose and angle is compared to the projected values, higher values indicate fault in right actuator, lower values indicate fault in left actuator, otherwise is considered a false positive.

Helper's Plans

When the Faulty robot fails to isolate the source of a fault, it asks for help, broadcasting a help request in the network. This situation results in the Helper robot receiving a request to help Faulty, and the two robots trying to diagnose the problem interacting between them, using the plans presented in the behavior trees shown in Figure 6.9 and Figure 6.10. The next steps refer to Helper's behavior tree, shown in Figure 6.10.

When the Helper robot receives a request for help from Faulty (`helpMe`), it first tries to detect the Faulty robot by slowly rotating on its axis until both colored cylinders from the Faulty robot are close to the center of its field of vision (`findRobot`). Once Faulty was detected and Helper

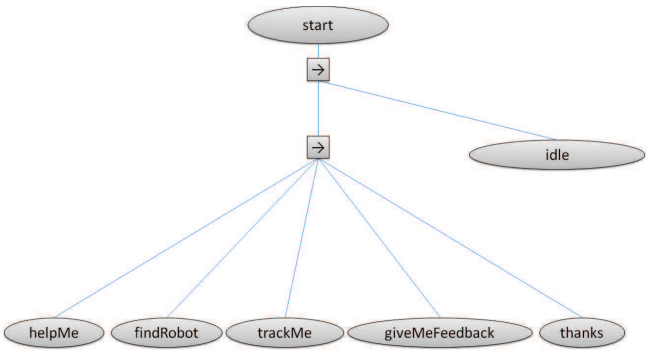


Figure 6.10: Helper's behavior tree

is positioned, the latter informs Faulty that it has found it. Upon receiving confirmation that it has been detected, Faulty in turn asks Helper to keep tracking it while it moves (`trackMe`). Helper then proceeds to track Faulty's movement by telling it to move, and observing Faulty's behavior. When Faulty has completed its movement, it requests Helper's perceptions about its position, pose, and depth (`giveMeFeedback`). Once the diagnosis has been completed, Faulty sends a message to Helper to let it know that the cooperative diagnosis procedure is finished, so Helper can reset the information about Faulty and return to its tasks (`thanks`).

7. RESULTS

This chapter evaluates the fault scenarios described in chapter 6 in terms of latency to complete the diagnosis steps, which are: the fault detection step, the robot location step, and the fault diagnosis step.

The robots are positioned side by side and separated by a distance of 1.5 meters in order to give *Faulty* enough space to move according to the injected fault, and to create in *Helper* the need for movement to find *Faulty*.

The faults are injected before executing the plan of the agent associated with the robot *Faulty*, in order to keep the metrics consistent to assess / compare the latency in the system.

Several runs were performed for each one of the proposed fault scenarios. The tests were recorded during theirs executions and the elapsed time for each step was extracted based on the logs.

The following sections present the results for each modeled faults, such as, fault in both actuators, fault in encoders, fault in left actuator and fault in right actuator.

7.1 Fault in both actuators

In this fault scenario, both actuators have faults. *Faulty* tries to move and, after receiving a fault event with reason *"no_progress"*, asks for help. *Helper* should locate and track *Faulty*, giving *Faulty* some feedback. Based on the feedback, *Faulty* should identify a fault in both actuators since both initial and final positions are the same.

The injected fault was detected, *Faulty* was located by *Helper* and the cooperative diagnosis has successfully been done in all runs (Figure 7.2). Table 7.1 shows the value for each step of all runs, illustrated in Figure 7.1. The mean time is 22.42 seconds and the standard deviation is 0.78 seconds, which corresponds to less than 4% of the mean time.

	Fault detection step (sec)	Robot location step (sec)	Diagnosis step (sec)	Total (sec)
Run 1	4.3	14.7	4.4	23.4
Run 2	4.0	14.2	4.7	22.9
Run 3	4.5	13.8	4.2	22.5
Run 4	4.1	12.4	4.9	21.4
Run 5	4.5	13.1	4.4	22.0

Table 7.1: Latency in the cooperative diagnosis of faults in both actuators.

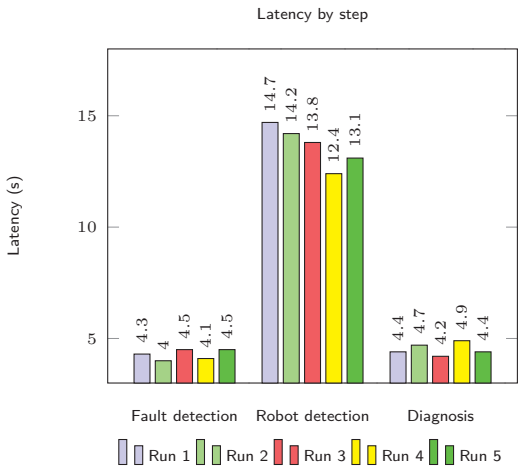


Figure 7.1 – *Fault in both actuators - latency by step.*

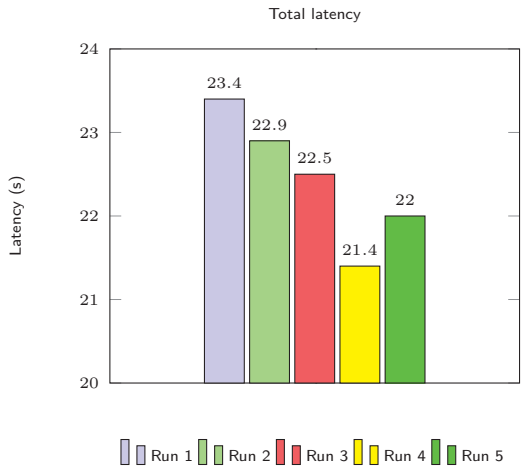


Figure 7.2 – *Fault in both actuators - total latency.*

7.2 Fault in the Encoders

On this fault scenario, *Faulty* tries to move and, as the previous fault scenario described in Section 7.1, a fault event with reason *"no_progress"* is received. But differently from the previous fault scenario, the actuators worked properly, but the encoders do not detect any movement in the wheels due to a fault, making them report always the same distance traveled. Thus, *Faulty* asks for help. *Helper* should locate and track *Faulty*, giving *Faulty* some feedback. Based on the feedback, *Faulty* should identify a fault in encoders once the final position is different from the initial position.

The injected fault was successfully detected in all runs, and *Faulty* robot was detected successfully in 85%. During run number 5, the robot detection step was aborted after *Helper* completed a turn of 360°, approximately 62.3 seconds, without detecting *Faulty*. Table 7.2 shows the value for each step of all runs. From the successful runs, the mean time is 25.55 seconds and the standard deviation is 10.97 seconds, which corresponds to approximately 43% of the mean time.

	Fault detection step (sec)	Robot location step (sec)	Diagnosis step (sec)	Total (sec)
Run 1	7.1	7.4	7.0	21.5
Run 2	6.7	7.0	6.1	19.8
Run 3	6.6	8.7	5.7	21.0
Run 4	7.3	8.5	5.8	21.6
Run 5	7.3	62.3	∞	∞
Run 6	7.7	35.1	5.1	47.9
Run 7	6.7	8.8	5.9	21.5

Table 7.2: Latency in the diagnosis of encoder faults.

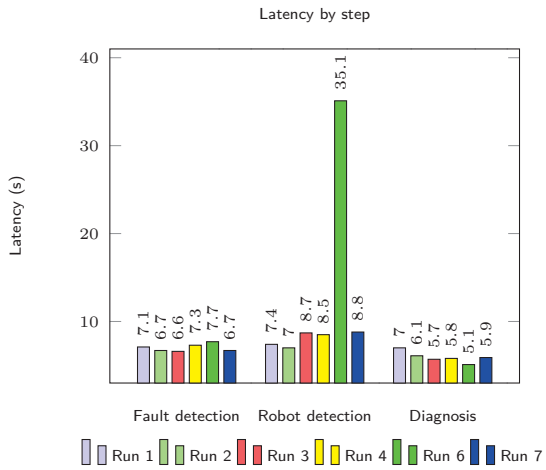


Figure 7.3 – *Fault in encoders - latency by step.*

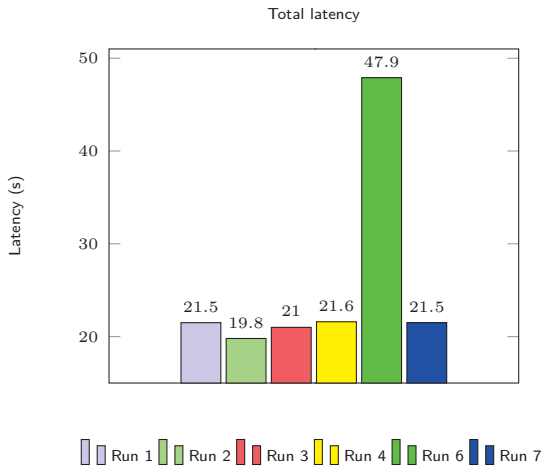


Figure 7.4 – *Fault in encoders - total latency.*

The high standard deviation is due to the fact that *Helper*, in run number 6, after detecting robot *Faulty*, required repositioning and, during the process of repositioning, *Helper* experienced some interaction faults. When *Faulty* is found, *Helper* tries to reposition itself in order to have *Faulty* at the center of its visual field. When the process of repositioning occurred, *Faulty* was not perceived in the center of image and, as a consequence, *Helper*'s belief base was not updated. Thus, despite *Faulty*'s being at the center of the visual field, *Helper* continued believing that *Faulty* was at its right side and kept moving until *Faulty* was perceived and *Helper*'s belief base updated. *Helper*, then, realized that *Faulty* was on its left side and it needed to reposition itself again. All the process of dealing with interaction faults during the *robot detection* step increased the time needed for *Helper* to get ready to help *Faulty*, as illustrated in Figures 7.3 and 7.4.

In run number 5, the interaction fault happened in the robot detection. During the time which *Faulty* was within the *Helper's* field of view, *Faulty* was not perceived or *Helper* had an incomplete perception of *Faulty*, i.e., the colored cylinders which identify *Faulty* were not detected in the same image frame. Thus, *Helper* was not able to help *Faulty*.

7.3 Fault in the Left Actuator

On this scenario, *Faulty*, after traveling a certain distance, receives a fault event with reason *"bad_move"*, which means the robot has taken an unplanned trajectory. *Faulty* asks for help to determine which actuator has a fault, or if encoders are informing wrong values. *Helper* should locate and track *Faulty*, giving it some feedback. Based on the feedback, *Faulty* should identify a fault in left actuator once its orientation, more precisely the angle of Z-axis, is smaller than the projected target angle.

Faulty was located by *Helper* and the cooperative diagnosis was successfully done in all runs. Table 7.3 shows the value for each step of all runs. The mean time is 30.25 seconds and the

standard deviation is 1.52 seconds, which corresponds to 5% of the mean time. Besides the fault being detected in all runs, in run number 3 the diagnosis was based on a perception of a previous location of *Faulty* due to occlusion of the orange cylinder. In run number 4, the diagnosis also occurred based on a perception of a previous location of *Faulty*, due to the impossibility of depth extraction from the point cloud and, for consequence, pose estimation when *Faulty* reached the target position. Figure 7.6 shows that issues do not have any impact in the latency and do not compromise the fault diagnosis.

Similar situations happened in both cases where the diagnosis was based on perceptions that did not represent the current location of *Faulty*. After *Faulty* was positioned on the right side of *Helper* and *Faulty* had a fault in the left actuator, *Faulty* performed a soft turn to the left side, positioning itself in front of *Helper*. *Faulty*'s pose, close to 90°in relation to *Helper*, caused the occlusion of the orange cylinder, preventing *Helper* from retrieving its depth.

	Fault detection step (sec)	Robot location step (sec)	Diagnosis step (sec)	Total (sec)
Run 1	18.5	0.9	10.3	29.7
Run 2	19.2	1.6	11.3	32.1
Run 3	18.1	2.4	10.5	31.0
Run 4	17.8	1.4	9.4	28.6

Table 7.3: Latency in the diagnosis of faults in the left actuator.

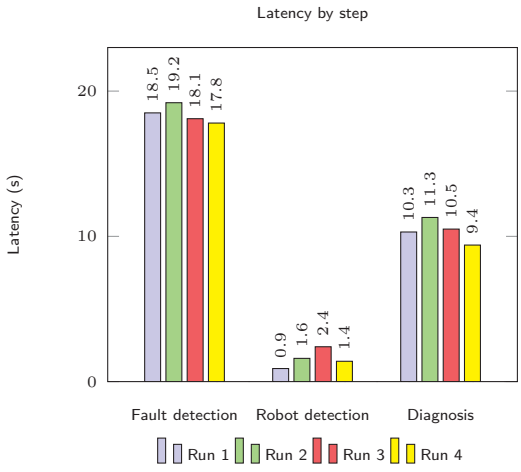


Figure 7.5 – *Fault in the left actuator - latency by step.*

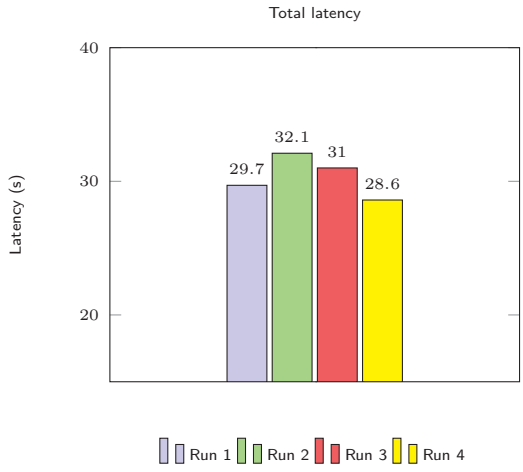


Figure 7.6 – *Fault in the left actuator - total latency.*

This scenario shows high latencies for the fault detection step, as illustrated in Figure 7.5, because *Faulty* moves slowly due to the actuator fault. Low latencies in the robot detection step are a consequence of *Faulty*'s movement towards the wrong direction, positioning itself in front of *Helper*. When the robot detection steps began, *Helper* had already perceived *Faulty*.

7.4 Fault in the Right Actuator

This scenario is similar to the previous one presented in Section 7.3. Despite similarities related to the behavior of the robots and fault characteristics, instead of positioning itself in front of *Helper*, *Faulty* performs a soft turn to the right side, moving away from *Helper*.

The injected fault was detected, *Faulty* robot was located by *Helper*, and the cooperative diagnosis was successfully done in all runs. Table 7.4 shows the value for each step of all runs. The mean time is 40.73 seconds and the standard deviation is 1.60 seconds, corresponding to less than 4% of mean time. Besides the fault being detected in all runs, in run number 3 the diagnosis was based on a perception of the previous location of *Faulty*, due to the non-detection of the orange cylinder caused by lighting issues. Figure 7.8, shows that the issue did not have any impact on the latency and did not compromise the fault diagnosis.

	Fault detection step (sec)	Robot location step (sec)	Diagnosis step (sec)	Total (sec)
Run 1	17.6	10.6	11.3	39.5
Run 2	18.9	9.2	14.2	42.3
Run 3	16.9	11.9	13.1	41.9
Run 4	15.6	10.8	12.8	39.2

Table 7.4: Latency in the diagnosis of faults in the right actuator.

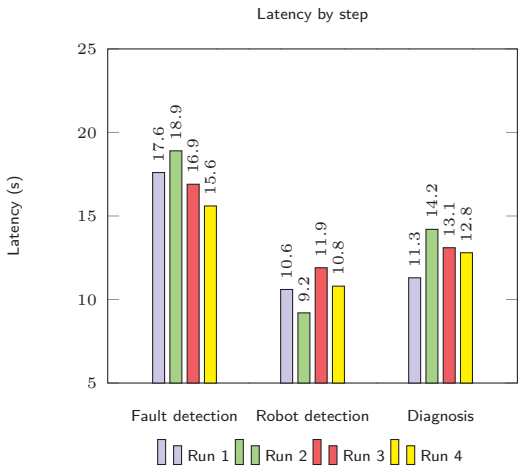


Figure 7.7 – *Fault in the right actuator - latency by step.*

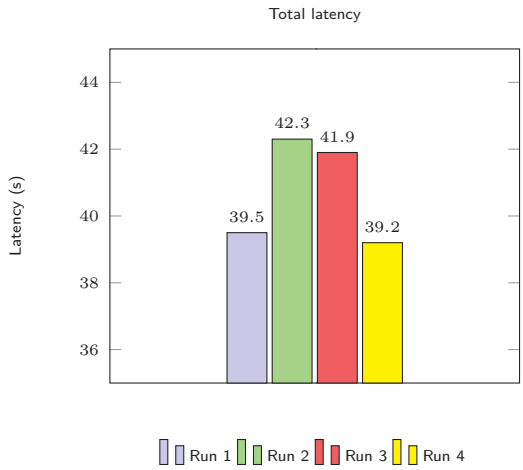


Figure 7.8 – *Fault in the right actuator - total latency.*

As the previous section, the fault detection step has high latency, as illustrated in Figure 7.7, due to a slow movement caused by the fault in the actuator. Differently from the previous scenario, the robot detection step has its latency time increased since *Helper* had to turn to locate *Faulty*.

7.5 Challenges

Since the beginning of this work we decided to work with real robots instead of simulation and we were (partially) aware of the challenges of evaluating even simple tasks in real and not-structured environments. This section describes some of these challenges experienced during the tests.

The dynamism and unpredictability of the real world is the motivation for using real robots instead of simulators. The use of simulation applied to fault tolerance boils down to the known cases or those planned by the researcher. As seen in this work, in the real world, even the planned scenarios presents unpredictable situations which should be considered in the research.

In the case of this work, computer vision is used to identify the robots and it is directly influenced by the environment, for example, variation in the light intensity.

The incidence of light on an observable object can cause it to be undetectable. In the implementation of the scenarios of this work, a classroom was used as a testing environment given that it is possible to prevent external light from entering the environment. Still, during the process of detection of the cylinders, false positives occurred from the incidence of light (or absence of) at certain points of the curtains or walls.

The unpredictability of the environment also influenced the tests. A burnt lamp changed the classroom lighting. The deficient lighting has projected shadows that hindered perception of colored cylinders.

There is also the challenges related to hardware details that are ignored in simulators. For example, Kinect cannot extract the depth from a point when there is a discrepancy between the depth of nearby points or when a sudden movement is performed.

The situations mentioned in this section are not experienced in simulators, but they can impact directly in the results of the research.

7.6 Discussion of Results and Applications

The cooperative diagnosis method presented in this work is efficient in diagnosing the hardware faults defined in case studies when in the absence of some interaction faults. However, with the dynamism and unpredictability of the real world, interaction faults cannot be ignored. In the faults scenarios presented in Chapter 6, it was experienced two sort of interaction faults: colored cylinder occlusion and non-observation of a colored cylinder. The cylinder occlusion happened when the observed robot placed itself in front of the observer robot and its orientation in Z-Axis is close to 90°. The non-observation of a colored cylinder has happened due to light incidence on cylinder caused changes in brightness of the other one because of its shadow, leaving the resulting color out of the range to OpenCV's *InRange* function. Treat this faults is considered as future work.

Not considering the runs where occurred interaction faults, the model remained stable in the execution of fault scenarios, comparing the running times of each step within each scenario. The variation in the running time of the steps is assigned to factors such as connection to the ROS master, inertial movement of the robots, network delays, etc.

8. CONCLUSION

This work presented Rason, a standardized interface to integrate ROS robotic framework and Jason multi-agent system. The integration allows to describe simple or complex tasks to robots through high-level plans using agent language. Furthermore, it enables the cooperation between robots once Jason can run distributed and it provides a well-defined standard of communication between agents.

It is considered as contribution of this work, the possibility of increasing code reusability through high-level plans and specialized executive modules (decomposers and/or synthesizers). Since it is needed to use an agent plan in a different robot, the robot just needs to have executives modules suitable for its hardware characteristics and which provide the necessary resources to the plan. On the other hand, plans for different purposes can share actions and/or perceptions provided by an executive module.

The separation of responsibilities provided by the proposed architecture also contributes to robustness and efficiency of the system once modules and plans may be replaced without any major change in the system, e.g. the replacement of the current computer vision module by another module more efficient would be transparent. It is important to mention that no changes in Rason code are required to have different modules, respected the rules described in Sections 5.3 and 5.4.

Once the core classes are done, modules of synthesis and decomposition can be created according to the hardware available in the robot, and these new modules can be shared among several different projects in which the same behaviors / perceptions are expected. Moreover, taking in consideration the abstraction of hardware details in Jason plans, new generic plans which cover specific situations can be developed, extended and shared, increasing modularity and reusability through the combination between agent's plans and packages of synthesis and/or decomposition.

Regarding the proposed case study, once the results presented in Chapter 7 show a success rate of 95% (19 out of 20 runs), it is considered viable the development of better plans to diagnose faults as well as new plans to perform a compensation of the workload, to delegate tasks or even reconsider plans and goals. Thus, it is possible to increase robustness and fault tolerance in the system.

This work also contributes to the academic community in the sense of providing the use of real robots instead of simulators, moving closer to real situations where unexpected events happens.

8.1 Future Work

The *RasonNode* class uses a manual handling of group of perceptions to avoid perception overwriting. Synthesizers *rosnodes* should inform a source of a perception list to prevent from overwriting. The development of an "intelligent management", able to perform a "perception filtering", and managing perceptions updates and new perceptions is considered as future work.

Another improvement is to propose the development of perception managers in form of plug-in to allow the use of different strategies of management. Such improvement would allow evaluation and use of different strategies as well as the creation of new strategies without any code modification in Rason classes.

In the case study presented in Chapter 6, a generic plan of observation of robots has been used in order to show that, even through the minimalist development of a single generic plan to treat those faults, cooperation at a high level is possible. Although the plan had successfully achieved its objectives, it can be extended to reduce untreated faults, in this case, the interaction faults experienced during the execution of the fault scenarios. For instance, the repositioning of the observer robot immediately behind the robot that requests help could reduce occurrences of occlusions or even perception losses due to distance limitations.

Considering the improvement of the synthesizer *rosnode* to deal with self-occlusion of the cylinders as a future work, it is suggested the use of a third colored cylinder. The approach solves problems of self-occlusion once at least two cylinders are always visible.

Diagnosis is just a step of a fault tolerant architecture. With such software abstraction it is possible to build a complete fault tolerant system with system reconfiguration and more advanced multi-robot collaboration through control loops as MAPE-K [18]. MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) is control loop reference model for self-repairing systems where the use of Rason could increase the reusability of control elements.

BIBLIOGRAPHY

[1] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. "Basic concepts and taxonomy of dependable and secure computing", *Dependable and Secure Computing, IEEE Transactions on*, vol. 1–1, 2004, pp. 11–33.

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

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

[4] Boronat Roselló, E. "Rosapl: towards a heterogeneous multi-robot system and human interaction framework", Master's Thesis, Universitat Politècnica de Catalunya, 2014.

[5] Carlson, J.; Murphy, R. R. "How ugvs physically fail in the field", *Robotics, IEEE Transactions on*, vol. 21–3, 2005, pp. 423–437.

[6] Carrasco, R. A.; Núñez, F.; Cipriano, A. "Fault detection and isolation in cooperative mobile robots using multilayer architecture and dynamic observers", *Robotica*, vol. 29–4, 2011, pp. 555–562.

[7] Corp, M. "Drivers and libraries for the xbox kinect device on windows, linux, and os x". https://github.com/OpenKinect/libfreenect, Dec 2014.

[8] Crestani, D.; Godary-Dejean, K.; et al.. "Fault tolerance in control architectures for mobile robots: Fantasy or reality?" In: CAR 2012: 7th National Conference on Control Architectures of Robots, 2012.

[9] D'Amaro, P. "Baderna eletromagnética", *Revista Superinteressante*, vol. 078, Mar 1994.

[10] Dix, J.; Fisher, M. "Where logic and agents meet", *Annals of Mathematics and Artificial Intelligence*, vol. 61–1, 2011, pp. 15–28.

[11] Elkady, A.; Sobh, T. "Robotics middleware: A comprehensive literature survey and attribute-based bibliography", *Journal of Robotics*, vol. 2012, 2012.

[12] FIPA. "Agent communication language specifications". http://www.fipa.org, 2014.

[13] Fletcher, L.; Teller, S.; Olson, E.; Moore, D.; Kuwata, Y.; How, J.; Leonard, J.; Miller, I.; Campbell, M.; Huttenlocher, D.; et al.. "The mit–cornell collision and why it happened", *Journal of Field Robotics*, vol. 25–10, 2008, pp. 775–807.

[14] 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.

[15] Goebel, R. P. "ROS By Example Volume 2 - Hydro". 2014, vol. 2.

[16] Han, J.; Shao, L.; Xu, D.; Shotton, J. "Enhanced computer vision with microsoft kinect sensor: A review", *Cybernetics, IEEE Transactions on*, vol. 43–5, 2013, pp. 1318–1334.

[17] Iñigo-Blasco, P.; Diaz-del Rio, F.; Romero-Ternero, M. C.; Cagigas-Muñiz, D.; Vicente-Diaz, S. "Robotics software frameworks for multi-agent robotic systems development", *Robotics and Autonomous Systems*, vol. 60–6, 2012, pp. 803–821.

[18] Kephart, J.; Kephart, J.; Chess, D.; Boutilier, C.; Das, R.; Kephart, J. O.; Walsh, W. E. "An architectural blueprint for autonomic computing", *IEEE internet computing*, vol. 18–21, 2007.

[19] Kohler, D.; Conley, K. "rosjava–an implementation of ros in pure java with android support". https://code.google.com/p/rosjava/, Dec 2014.

[20] Kramer, J.; Scheutz, M. "Development environments for autonomous mobile robots: A survey", *Autonomous Robots*, vol. 22–2, 2007, pp. 101–132.

[21] Lussier, B.; Chatila, R.; Ingrand, F.; Killijian, M.-O.; Powell, D. "On fault tolerance and robustness in autonomous systems". In: Proceedings of the 3rd IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, 2004.

[22] Michel, O. "Webots: Professional mobile robot simulation", *International Journal of Advanced Robotic Systems*, vol. 1–1, 2004, pp. 39–42.

[23] Mordenti, A.; Ricci, A.; Santi, D. I. A. "Programming robots with an agent-oriented bdi-based control architecture: Explorations using the jaca and webots platforms", Bachelor's thesis, Università di Bologna, 2012.

[24] Pokahr, A.; Braubach, L.; Lamersdorf, W. "Jadex: A bdi reasoning engine". In: *Multi-agent programming*, R. Bordini, M. D.; Seghrouchni, A. E. F. (Editors), USA: Springer Science+Business Media Inc., 2005, chap. 6, pp. 149–174.

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

[26] Rao, A. S. "Agentspeak (l): Bdi agents speak out in a logical computable language". In: *Agents breaking away: Proceedings of the seventh european workshop on modelling autonomous agentsin a multi-agent world.*, de Velde, W. V.; Perram, J. W. (Editors), Berlin, Heidelberg, New York, Germany: Springer–Verlag, 1996, vol. 1038, pp. 42–55.

[27] Rockel, S.; Klimentjew, D.; Zhang, J. "A multi-robot platform for mobile robots—a novel evaluation and development approach with multi-agent technology". In: Multisensor Fusion and Integration for Intelligent Systems (MFI), 2012 IEEE Conference on, 2012, pp. 470–477.

[28] Russell, S.; Norvig, P.; Intelligence, A. "A modern approach", *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, 1995.

[29] 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*, Seghrouchni, A. E. F.; Leite, J.; Torroni, P. (Editors), Springer, 2011, pp. 95–114.

[30] Weber, T.; Jansch-Pôrto, I.; Weber, R. "Fundamentos de tolerância a falhas", *SBC/UFES*, 1990.

[31] Wooldridge, M. "An introduction to multiagent systems". John Wiley & Sons, 2009.

[32] Wooldridge, M.; Jennings, N. R. "Intelligent agents: Theory and practice", *The knowledge engineering review*, vol. 10–02, 1995, pp. 115–152.

[33] YUJINROBOT. "iclebo kobuki". http://kobuki.yujinrobot.com/, Dec 2014.

APPENDIX A – AGENT'S PLANS

Listings APPENDIX A.1 and APPENDIX A.2 show the plans for *Faulty* and *Helper*, respectively.

Faulty's plans

```
// Agent robot in project robot.mas2j

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start <- .wait(4000); !goToDestination.

+!goToDestination
  <- .print("Moving to Destination");
     move(1);
     move(-1);
     .print("Arriving at destination").

-!goToDestination[R1 | Rest]
   : triedAlone
  <- .print("Cannot solve problem alone, asking for help.");
     +reason(R1);
     .broadcast(achieve,helpme).

-!goToDestination[R1 | Rest]
  <- .print("something went wrong...",R1);
     !diagnoseAlone([R1| Rest]);
     !goToDestination.

+!diagnoseAlone(Reasons)
   : .member(no_progress,Reasons)
   | .member(bad_move,Reasons)
  <- .print("Running local diagnosis");
     .print("diagnose alone has failed");
     +triedAlone;
     .print("Finished diagnosis").

//Plans to help diagnose
+!tryMove[source(R)]
  <- .print("Trying to bit for ",R);
     move(0.3);
     .print("It was just a false positive.");
     -reason(_);
     -triedAlone;
     .send(R,achieve,thanks).

-!tryMove[source(R)]
  <- .print(" requesting some feedback from ",R);
     .send(R,achieve,givemeFeedback).

+found(Me)[source(R)]
   : .my_name(Me)
  <- .print(R," found me, asking to track me");
     .send(R,achieve,trackMe).

+!feedback(initPos(Deg,Depth,Yaw),currPos(Deg1,Depth1,Yaw1))[source(R)]
    : reason(no_progress)
    & ( not (Depth == Depth1) | not (Deg == Deg1) )
  <- .print(Helper," has reported as initial info( pos ",Deg," degrees, depth ",Depth," meters and yaw ",Yaw," degrees )");
       .print("and current info( pos ",Deg1," degrees, depth ",Depth1," meters and yaw",Yaw1," degrees )");
     .print("fault in odometry system, thanks ", R);
     .send(R,achieve,thanks).

+!feedback(initPos(Deg,Depth,Yaw),currPos(Deg1,Depth1,Yaw1))[source(R)]
    : reason(no_progress)
    & Depth == Depth1
    & Deg   == Deg1
  <- .print(Helper," has reported as initial info( pos ",Deg," degrees, depth ",Depth," meters and yaw ",Yaw," degrees )");
```

```
69          .print("and current info( pos ",Deg1," degrees, depth ",Depth1," meters and yaw",Yaw1," degrees )");
70        .print("fault in both actuators, thanks ",R);
71        .send(R,achieve,thanks).
72
73  +!feedback(initPos(Deg,Depth,Yaw),currPos(Deg1,Depth1,Yaw1))[source(R)]
74      : reason(bad_move)
75      & math.abs(Yaw1 - (Yaw + (Deg1 - Deg))) < 8
76    <- .print(Helper," has reported as initial info( pos ",Deg," degrees, depth ",Depth," meters and yaw ",Yaw," degrees )");
77          .print("and current info( pos ",Deg1," degrees, depth ",Depth1," meters and yaw",Yaw1," degrees )");
78        .print("it was a false positive for bad_move, thanks ",R);
79        .send(R,achieve,thanks).
80
81  +!feedback(initPos(Deg,Depth,Yaw),currPos(Deg1,Depth1,Yaw1))[source(R)]
82      : reason(bad_move)
83      & ((Yaw1 - (Yaw + (Deg1 - Deg))) > 8)
84    <- .print(Helper," has reported as initial info( pos ",Deg," degrees, depth ",Depth," meters and yaw ",Yaw," degrees )");
85          .print("and current info( pos ",Deg1," degrees, depth ",Depth1," meters and yaw",Yaw1," degrees )");
86        .print("right actuator has problems, thanks ",R);
87        .send(R,achieve,thanks).
88
89  +!feedback(initPos(Deg,Depth,Yaw),currPos(Deg1,Depth1,Yaw1))[source(R)]
90      : reason(bad_move)
91      & ((Yaw1 - (Yaw + (Deg1 - Deg))) < -8 )
92    <- .print(Helper," has reported as initial info( pos ",Deg," degrees, depth ",Depth," meters and yaw ",Yaw," degrees )");
93          .print("and current info( pos ",Deg1," degrees, depth ",Depth1," meters and yaw",Yaw1," degrees )");
94        .print("left actuator has problems, thanks ",R);
95        .send(R,achieve,thanks).
```

Listing APPENDIX A.1: Faulty's plan.

- Line 7 - This line defines the initial goal *start*;
- Line 11 - This line defines the plan to reach *start*. The internal action *wait* is necessary to give some time so Rason can establish connections to ROS Master. *goToDestination* describes the actions to reach the goal;

  Lines 13 to 36 describe the actions to reach the goal and the fault plans for that plan.

- Line 13 to 17 - *goToDestination* - The actions to reach the goal. Basically moving one meter ahead and one meter back. It is a simple task, but enough to validate the experiments when a fault is injected.
- Lines 19 to 23 - The fault plan for *goToDestination* when an action has failed and the agent has already tried to diagnose it alone (agent has in the belief base *triedAlone*). The agent sends a help request through a broadcast message and waits for any response;
- Lines 25 to 28 - The fault plan for *goToDestination* when an action failed and the agent wants to try to diagnose it alone.
- Lines 30 to 36 - If the reason for fault is *"no_progress"* or *"bad_move"*, the agent tries to diagnose it alone. Since the ambiguity in the fault scenarios, this is not possible. Thus, it just appends the belief that the robot has tried to do it by adding *triedAlone* to belief base;

  Lines 38 to 54 are plans to help diagnose.

- Lines 39 to 45 - After being detected by *Helper*, *Faulty* tries to move in order to be tracked by *Helper*. It is expected that the *move* action also triggers a fault event, since Faulty believes

it has a defective device. Once a fault plan for *tryMove* is selected, *Faulty* proceeds to fault diagnosis. However, if *move* action is successful, the fault event is considered a false positive and the beliefs related to the fault are removed from the base;

- Lines 47 to 49 - Fault plan for *tryMove* plan. This plan is selected during the diagnosis process, after the agent tries to move unsuccessfully. This plan requests feedback from *Helper* about the observed properties: angle, pose and depth;
- Lines 50 to 54 - When *Helper* finds *Faulty*, *Helper* reports its availability to help, then, *Faulty* asks to be tracked;

Lines 56 to 95 are the feedbacks from *Helper* to cooperate in the *Faulty*'s diagnosis process.

- Lines 56 to 62 - Once *Faulty* received a feedback from Helper, and the fault reason is *"no_progress"*, whether the current depth is different from the initial depth or the current angle is different from the initial angle, the encoders are pointed as source of fault once the changes observed in the parameters *depth* and *angle* characterize a robot movement;
- Lines 64 to 71 - When the fault reason is *'no_progress"* and no changes are detected in parameters *depth* and *angle*, actuators are considered as the source of fault;
- Lines 73 to 79 - When the fault reason is *"bad_move"* and the current orientation and the projected target orientation have an absolute difference less than 7°, it is considered that the robot has moved within the trajectory and the fault event is considered as a false positive;
- Lines 81 to 87 - When the fault reason is *"bad_move"* and the difference between the current orientation and the projected target orientation value on Z-axis is greater than the initial, it is diagnosed as a right actuator fault;
- Lines 89 to 95 - When the fault reason is *"bad_move"* and the difference between the current orientation and the projected target orientation value on Z-axis is less than the initial, it is diagnosed as a left actuator fault;

Helper's plans

```
1  /* Initial beliefs and rules */
2  turnRate(5).
3  centerThreshold(7).
4  /* Initial goals */
5  !start.
6
7  /* Plans */
8
9  +!start : true <- .wait(4000); .print("ZZZzzz...").
10
11 +!helpme[source(R)]
12   <- +helping(R);
13      !findRobot(R,0);
14      .print("helper ready.");
15      .send(R,tell,found(R)).
16
17 +!findRobot(Robot,Turned)
```

```
18    : robot(Robot,Deg,Depth,Yaw)
19    & centerThreshold(DT)
20    & ( math.abs(Deg) >= DT )
21    <- .print("Found ",Robot, ", but I need to reposition myself...");
22       turn(Deg);
23       !findRobot(Robot,Turned+Deg).
24
25  +!findRobot(Robot,Turned)
26    : robot(Robot,Deg,Depth,Yaw)
27    & centerThreshold(DT)
28    & math.abs(Deg) < DT
29    <- .print("Found ",Robot);
30       +foundRobot(Robot,Deg,Depth,Yaw).
31
32  +!findRobot(Robot,Turned)
33    : turnRate(Rate)
34    & centerThreshold(DT)
35    <- .print("Finding ",Robot," turning ",Rate," degrees, turned ",Turned," so far. Threshold is ",DT);
36       turn(Rate);
37       !findRobot(Robot,Turned+Rate).
38
39  +!trackMe[source(R)]
40    : helping(R)
41    & foundRobot(R,Deg,Depth,Yaw)
42    <-  .print("I'm ready.");
43        .send(R,achieve,tryMove).
44
45  +!givemeFeedback[source(R)]
46    : helping(R)
47    & foundRobot(R,Deg,Depth,Yaw)
48    & robot(R,Deg1,Depth1,Yaw1)
49    <- .print("reporting to ",R," -> ",Deg," ",Depth," ",Yaw," / ",Deg1," ",Depth1," ",Yaw1);
50       .send(R,achieve,feedback(initPos(Deg,Depth,Yaw),currPos(Deg1,Depth1,Yaw1))).
51
52  +!thanks[source(R)]
53    : helping(R)
54    <- .print(R," said 'thanks'...");
55       .print("You're welcome, ",R);
56       -helping(R);
57       -foundRobot(R,_,_,_).
58
59  +robot(Robot,Deg,Depth,Yaw)
60    <- -+ robot(Robot,Deg,Depth,Yaw);
```

Listing APPENDIX A.2: Helper's plan.

- Line 2 - The line defines the turn rate when *helper* is looking for *Faulty*, by defining that initial belief;
- Line 3 - The line defines a threshold to consider *Faulty* positioned in the center of the field of view, by defining that initial belief;
- Line 5 - The initial goal *start*;
- Line 9 - The line defines the plan to reach *start*. The internal action *wait* is necessary to give some time so Rason can establish connections to ROS Master. *Helper* is waiting for help requests but it could be doing any other task;
- Lines 11 to 15 - When a help request is received from another agent, *Helper* starts looking for the agent that requested help and, when the agent is found, notifies it;
- Lines 17 to 24 - When looking for *Faulty*, if *Helper* perceives *Faulty* and it has the belief that *Faulty* is located outside the central region of its field of view, *Helper* repositions itself based on the angle from which *Faulty* was perceived;

- Lines 26 to 31 - When looking for *Faulty*, if *Helper* perceives *Faulty* inside the central region, it believes that it is helping *Faulty* and enables the plan *helpme* (Line 11) to notify *Faulty*;
- Lines 33 to 38 - When looking for *Faulty*, if *Faulty* has not been found, it turns *Rate* degrees and reinserts the intention with *Turned* value of *Turned+Rate*;
- Lines 39 to 43 - If *Faulty* wants to be tracked and *Helper* has an initial position, it ask *Faulty* to move;
- Lines 45 to 50 - When *Faulty* requests some feedback, *Helper* reports the initial and the current position;
- Lines 52 to 57 - Once *Faulty* has finished the diagnosis, it removes the beliefs related to the cooperative diagnosis from the base;
- Lines 59 to 60 - Updates the belief about the *Faulty* when new perceptions come from the synthesizer.