Alma Mater Studiorum
Università di Bologna

II Facoltà di Ingegneria

Corso di laurea magistrale in Ingegneria Informatica

Programming Robots with an Agent-Oriented BDI-based Control Architecture: Explorations using the JaCa and Webots platforms

Tesi in:
Sistemi Multi Agente

Candidato:
Andrea Mordenti

Relatore:
Prof. Alessandro Ricci

Correlatore:
Dott. Ing. Andrea Santi

Anno Accademico 2011/2012 - Sessione II

Keywords

Domestic Robots
BDI architecture
Jason
CArtAgO
A&A meta-model
Webots

To you,
who are looking after me from up there.

Contents

Introduction

What was a barrier few years ago, like the high costs of hardware (e.g GPS, infrared sensors, cameras and so on) has become nowadays a set of affordable devices. Thus, now, robot builders can add to robot's structure these devices with a reasonable expense.

Besides hardware aspects and its trend over time, software in robotics has covered a great distance too. It plays a main role since is used to tell a mechanical device (a robot) which are the tasks to perform and control its actions. It is worth pointing out that specifying the control logic at software level, provides more flexibility, in addition gives programmers an easy way to extend and change the robot behaviour.

Like in the '70s when Bill Gates and Paul Allen looked at the convergence of new technologies and dreamed about the day when computers would have become smaller and cheaper, hence in every home[8]. Thus, it is not so weird to imagine a future where autonomous devices become a integrated parts of our day-to-day lives. These devices are constantly the more powerful, cheaper and smaller, the more time passes. The robotics domain has experienced a shift from robots big and expensive -like the mainframes- to small and pervasives ones -personal computers. This shift can be interpreted as the transition from robots who compose assembly lines of automobile manufacturing or in military missions (e.g drones for US Army), to robots who perform companionship activities, physical assistance or cleaning tasks.

In this perspective, robot programming becomes a very important aspect, like programming stardard applications. That is, robots are considered as computer systems, not just electronic and mechanic devices. Considering that, this thesis will mainly focus on programming robotic control systems, that is a challenging tasks for today's programmers with even a basic under-

standing of robotics.

There is the need then to look up for models and architectures as well as languages **general purpose**, to design a robot. Being general purpose is a fundamental requirement for the robot's control architecture that we take into account to program robotics systems.

On the one side, different kind of general purpose well-suited architectures are provided by the literature (e.g deliberative, reactive, hybrid, behaviour-based) for programming robots. On the other side, traditional programming languages are typically used to implement concretely robot programs. However, such languages do not provide either abstractions or mechanisms to cope with critical aspects concerning robot programming. In particular, with regard to the interaction between the environment and the design of reactive and autonomous behaviour of the robot. Therefore there is a conceptual gap between the high-level specification of robot's behaviour and the implementation -and design- of the program.

This thesis aims at tackling this issue, taking into account an agent-based approach for programming robots. In particular, we are going to use the BDI (Belief, Desire, Intention) model, which directly encompasses all the necessary features to program a robot and help to cope with typical robotics issues.

With regard to the above considerations, this thesis will explore a method to design robot controllers by means of high level languages and architectures like Jason and CArtagO. In order to do the experiments and tests, the Webots robot simulator platform is used. The platform allows to create simulated worlds and robots, and provides proper API to develop the required bridge to integrate heterogeneous programming language and systems for controlling the robot.

A set of small but relevant programming examples is used to compare the different approaches just mentioned. The different investigations will deal both with standard and high level programming languages in order to fulfill the aimed goals.

Of course programming robots is a non-trivial task, but in this work we show how agent programming languages and technologies could be a promising approach for easing the development of articulated robot programs, im-

proving their modularity and readability, and reducing the gap between the design and implementation level. As a result, the contribution brought by this thesis concerns the possibility to do robot programming in a easier way.

Organization of the thesis

This paper is organized as follows:

Chapter 1 In the first chapter we are going to explain what a robot and robotics are concerned, especially tackling specific topics like autonomous robots and domestic robotics. Afterwards will be showed some basic control architectures that are used to be enforced for robotics purposes. Then, the chapter goes ahead with some considerations about programming systems, why robotics and its programming is so meaningful and which are its drawbacks. Moreover, to provide a complete thesis background we will conclude talking about languages and platforms at the state of the art for robotics as well as the Webots simulator.

Chapter 2 Describes what are agent programming languages and agent systems in order to understand how they can be used as a basic approach in programming robots. With special regards to Jason BDI agent architectures and languages, we will try to explain why that could be a meaningful way to program an autonomous system like a robot.

Chapter 3 Shows how to use Jason and a related framework like CArtAgO to describe robot controller architecture, that works over the Webots simulator.

Chapter 4 Presents a bunch of examples implemented with both the agent-oriented and the standard approach (written either in C). Starting from a high level description of the problem and the related strategy, ending up to the actual implementation of the strategy (in both ways).

Chapter 5 Here we will compare what turns out from the experimentations showed in the previous chapter. By evaluating the differences between the approaches, the modularity brought in by Jason and finally discussing roughly about the performance.

Finally, the last chapter discusses the limits of the proposed approach and the extensions that can be considered in future work. For example:

how and why it would be extremelly useful and interesting looking ahead towards a connection between the mentioned languages and AI techniques. Considering that, such work can brought to a robotics programming more automated.

Chapter 1

Background

How could we define a **robot**[1]?

Of course an unique, correct and neat definition doesn't exist because of the complexity of such wide branch of technology, in addition there is a lot of misunderstanding about this topic. But seeking among a great number of articles a robot can be defined clearly as

> *"an autonomous system which exists in the physical wolrd, is able to sense its environment and can act on it to achieve some goals[7]".*

Two part of the above definition tell us what we have to deal with, when we are going to face the robotics field. The autonomy means that a robot can perform desired activity in unstructured environments without continuous human guidance, moving either all or part of itself throughout the environment without human assistance and avoiding harmful situation for people or itself. The ability of sensing (to touch, to see, to hear etc, by means of a set of sensors) and acting on the environment (through some devices called actuators/effectors) means that the robot has sensors in order to get information from the world and respond according to them properly, taking actions to achieve what is desired or rather to achive some goals, this may be considered the "intelligent part" of a robot.

[1]The word robot was introduced by a czech writer named Karel Capek in his play *Rossum's Universal Robots* in 1921, it comes from the Slavic word robota, which is used to refer "forced labor".

So, once we have defined what a robot is, we can now define what **robotics** means and we think a well suited definition according to the robot's one can be

> *"Robotics is the branch of technology that deals with the design, construction, operation and application of robots and computer systems for their control, sensory feedback, and information processing[11]."*

One of the main area of robotics research is to enable the robot to cope with its environment whether this be on land, underwater, in the air, underground, or in space in order to implement a fully autonomous robot which may also learn or gain new capabilities like adjusting strategies for accomplishing its task(s) or adapting to changing surroundings.

1 Programming Robots

Programming robots means to implement the robot software as the coded set of commands that tell a mechanical autonomous device what task to perform and control its action. The robotics industry faces many of the same challenges that the personal computer business tackled about 30 years ago: robot companies have many problems regarding the standardization of robotic processors, moreover only a little part of programming code used in a machine that controls a robot, can be applied again to another one: in fact, whenever a programmer wants to build a new robot, he has to start from square one and program a new controller from the scratch. Programming robots is a non-trivial task indeed, even though many software systems and frameworks (see further) have been developed to make programming robots esier.

Early robot programming approaches used to rely on data flow based techniques which model the whole robot's lifecycle as a simple sequence of actions. However as time went by, some robot software aims at developing intelligent mechanical devices that should be able to react to different (and maybe unpredicted) events: so pre-programmed hardware that (may) include feedback control loops such that it can interact with the einvironemnt does not display actual intelligent.

Thus, we have to take into account some different kinds of features to program a robot properly:

- the typology/purpose of robot we are going to develop (industrial, domestic, military, totally autonomous etc.),
- the control architecture,
- the underlying programming system we choose (and the related language).

It is worth to mention the high importance of a relevant model (and simulation, as we will show) of what we are going to program in terms of the above points, as it's hard to enforce our control program on a real autonomous system, like a robot.

1.1 (Domestic)Robotics

The world's robot population has been strongly rising during the last years; today's amount of robot has likely reached ten milion of units because of are become more powerful, with more sensors, more intelligence and equipped by cheaper components (as we told previously). As a result they moved out of controlled industrial environments into uncontrolled service environments such as houses, hospitals, and workplaces where they perform different sorts activities ranging from delivery services to entertainment.

Some studies divide robots in two main categories, *industrial* and *service* robots. The former category includes welding systems, assembly manipulators and so on, that carry out heavy, expensive and several degrees of freedom of tasks. The latter category is divided in two subcategories: *professional service robots* like bomb disposal machine, surgical systems, chargo handler, milking robots, and *personal service robots* like vacuum cleaners, lawn mowers, several sorts of new generation toys and hobby kits.

The kind of autonomous mechanical devices we are going to analyze and then to program along with this thesis is the one regarding the domestic and, in general, personal service robots; such kind of robots is on the one hand interesting beacause is the far more widespread (as we can see from the bar chart in figure 1.1) and on the other hand we think these autonomous devices lie on a field closer to our interest an farther from industrial ones. In particular we are not going to consider that specific branch of robotics

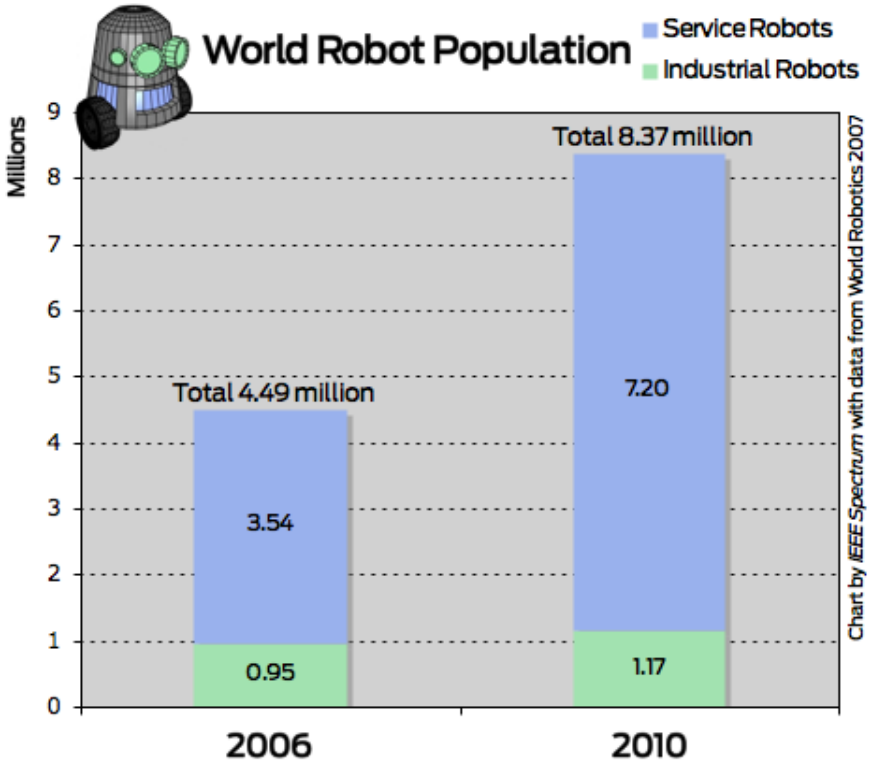

Figure 1.1: The service and industrial robots increase over four years.

concerning completely automated robots, which are based upon artificial intelligence techniques only, that allow to change and adapt dinamically their behaviours, even though we will show how the afterwards work could be extended towards such kind of programming.

1.2 Control Architecture

When we defined what a robot is, we talked about the set of devices that help a robot to perceive its physical world in order to get information about itself and the environment, the *sensors*. These devices are fundamental because through a right unit of sensors a robot knows its state, or rather can get a general notion of the current situation of the world in order to choose the relevant actions to enforce by means of another set of devices that a robot is equipped with, the actuators. The non-trivial operation that allow a robot to take information from sensors and select which is the best action to apply using the actuators is done by the **controller(s)**.

A controller plays the role of the "brain" and the "nervous system" of an autonomous system, it provides the software that makes the robot autonomous by using sensor inputs to decide what to do and then to control the actuators to execute those actions in order to pursued certain goals. A lot of controller categories exists, and we know that the simpliest one is the feedback control (data flow technique) (see figure) that is a nice way to write controllers for one-behaviour / simple-task robots which have no need of thinking, however nowadays robots are assembled to perform more complex tasks, whatever they are. Therefore putting together different simple feedback control loops is not the right answer to model a good controller, such task is not simple if we want to achieve a well-behave robot.

Thus, like *Design Patterns* employed in software engineering, we need some guiding principles and constraints for organizing the "brain" of our robot and then helping the designer to program its controller so that it behaves as desired, in a *language independent* way. Such choice will be taken despite of the programming language used to program a robot, in fact what matters is the control architecture used to implement the controller and hereafter we show some types of control.

Deliberative control

In this architecture there is some consideration in alternative courses of action before an action is taken, so deliberation could be defined as *thoughtfulness in decision and action* that involves the capacity to represent states referring to hypothetical past or future states or as yet unexecuted actions. So, deliberative control goes hand in hand with AI in order to solve very complex problems through *planning* operations.

Planning is the process of looking ahead at the outcomes of the possible actions, to realize strategies as a sequence of actions that will be executed by an intelligent agent in order to achieve a (set of) goal(s). However such operation -according to the complexity of the problem to solve- might has to take into account a huge amount of aspects; as result, that entail a cost in terms of time, memory and a possible lack of information. Indeed for non-trivial problems, the number of possible states that an agent in charge of

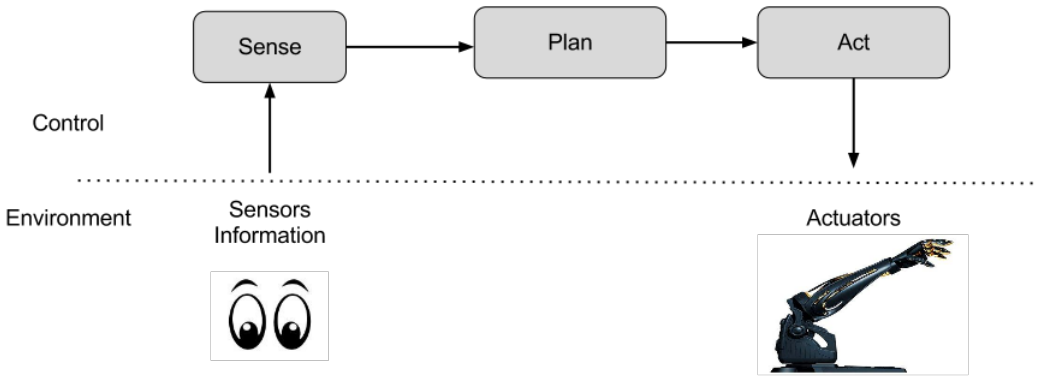

Figure 1.2: Simple deliberative architecture.

planning have to analyze becomes even extremelly large. So the longer it takes to plan, the slower a robot may become to enforce an action and it is not a good in case a robot situated within an uncertain environment must deal with hazardous situation. In addition, in such case a robot must get a considerable amount of updated information -that concern a plenty storage memory- in order to get consistent plan, but this is not always possible: if planning operation takes much time and it is situated inside an high changeble world it is likely to use outdated information and then to produce a not proper plan.

Of course there are some fields where pure deliberation is required -where there is no time pressure, in a static environment and low uncertainty in execution- even though they are rare and the have expanded the approach seen in fig 1.2.

Reactive Control

Reactive control is one of the most commonly used method to control a robot. These kinds of systems rely on a bunch of rules that connect sensory inputs (stimuli/conditions) to specific actions (responses/behaviours) neither with a representation of the external environment nor looking ahead to possible outcomes related to the application of an action, just without the need of

thinking.

A controller may select the suitable action to apply in different ways: on the one hand since data from sensors are continuous, to implement a correct reactive system we need an unique stimulus for given set of sensory inputs which trigger a unique action, this is what is called *mutually exclusive* condition. On the other hand is possible to have conflicts among the actions the controller may apply, so an action selection process is needed in order to decide the action to apply: a *command arbitration* looks like a selector that choose one action among those applicable behaviours whereas a *command fusion* combine multiple relevant behaviours ino a single one.

In reactive systems the action selection is challenging in case there are several rules and sensors state to check concurrently instead of in sequence: this means that first it has to support parallelism and then the underlying programming language must have the ability to execute several processes and commands in parallel[2].

Hybrid control

So far we have seen two kinds of control that are worlds apart: deliberative and reactive. The former is smart but bould become the slower the more complex is the problem, the latter is fast but less flexible. It is obvious consequence trying to take the best of both the approaches and put them together: that's the aim of hybrid control architectures. It is really complex to obtain though. A hybrid architecture typically consists of three components: a **reactive layer**, a **planner** and a **middle layer** that connects the previous two.

Although the first two components are known, the role of middle layer is blurry yet. Let's image we have a robot that executes a set of activities to reach some goals using both reactive rules and planning:

- what if it needs to start a certain activity in order to carry out a critical objective even though there is not yet a proper plan to enforce?
- what if the planning operation is blocked due to outdated data?

[2]This topic could be require a lot of considerations about the priority and/or the coordination among those rules

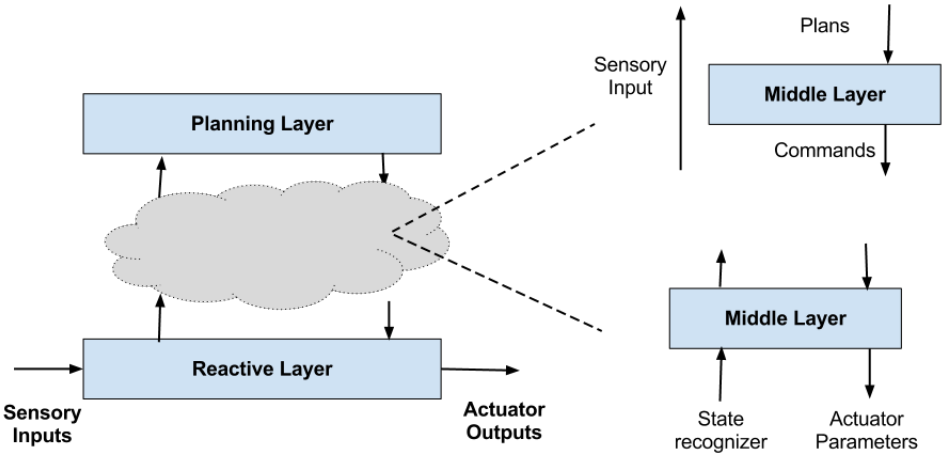

Figure 1.3: Some hybrid architecture fashions, from [7]

- what if a hazard comes out? Should it wait or produce a brand new plan?
- ....

When the reactive layer detects an unexpected situation that it cannot handle, probably it will inform the deliberative layer about such situation along with the related data. The deliberative layer take those information in order to create new suitable plans and provide the bottom layer the guidelines to act as correctly as possible.

That layer is useful to deal with above kinds of issue and its design (see fig.1.3) is the biggest challenge for hybrid architectures.

Behaviour-based control

This kind of control architecture is inspired by biological systems and aim to overcome those problems that turn out in the others approaches. That type of control are closer to the reactive one than to others, as a matter of fact a behaviour-based system is composed by **behaviour modules**. However, while behaviour based systems embody some properties of reactive systems and -usually- contain reactive components (behaviour modules) their computation is not limited to look-up and execution of simple functional mapping.

Behaviours can be employed to store varius forms of state and implement varius types of representation. As a result it neither have the limits of reactive systems nor employ a hybrid structure -with middle layers. To perform useful work in the real world we must have our robots do different things under different cirumstances: here enters the concept of **behaviour**.

Behaviours are more complex than actions, in fact while a system controlled by a sequence of simple actions like *go-ahead* or *turn-left*, this kind of control architecture uses time-extended behaviours like *reach-destination*, *avoid-obstacle* or *follow-the-light*. Such behaviours unlike actions, are not instantaneous and aim at achieve and/or mantain a particular state. Behaviours modules are executed in parallel or concurrently, is activated in response to incoming sensory inputs and/or outputs from another behaviour and it can also be incrementally added to the system in order to achieve a more skilled system. Given the last considerations the concept of behaviour could result similar to the concept of reactive rule, but the latter can be used with the purpose of obtaining more interesting results because:

"Behaviours are more expressive (more can be done with them) than simple reactive rules are."[7]

Since it is high probable the controller has to tackle concurrent execution of different behaviours, an arbitraton component is needed to maintains the execution correct at any time, even if there is competition among behaviours for rescources. The simpliest and commonly used in behaviour-based systems are fixed-priority arbiter[3][12] in which a constant and unique numerical priority is assigned to each behaviour, obviously then two or more behaviours conflicts the highest-priority behaviour is undertaken.

For example a vacuum cleaner robot that is wandering over a floor and have the ability to recharge its battery automatically thanks to a sensor that continuously check the battery level. The *wander behaviour* involves to move throughout the house -cleaning each room. When triggered, the charge behaviour issues motor commands that drive the robot toward the charging device. But what if the *charge-home behaviour* and the *wander behaviour*

[3]Of course it is not the only method to combine conflicting behaviours such as variable priority, subsumption architecture, motor schema, least commitment arbitration, etc.

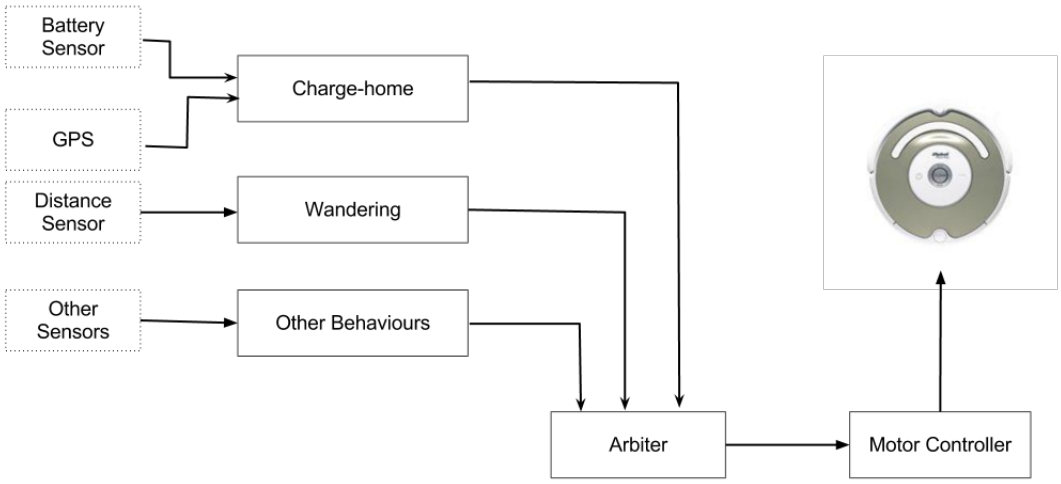

Figure 1.4: Behaviour-based system with potentially conflicting behaviours for robot's motor.

issue two different motor commands? For more complex case studies we need to take into account several aspects to determine how to resolve such conflict; in this case when the batteries are about to be exhausted it is more important that the robot head to the charging point othewise all the others tasks cannot be carried out. In fig1.4 we show a simple fixed-priority arbitration scheme.

BDI architecture

See Chapter 2

What we choose among the above categories will affect the subsequent steps in robot designing, therefore to program robot's control we must consider different things that are meaningful to select which architectures employ coherently such as:

- ♢ *is there a lot of sensor/actuator noise?*
- ♢ *is the environment static or dynamic?*
- ♢ *does the robot sense all the information it needs?*
- ♢ *how fast do thing happens? do all components run at the same speed?*

♢ *which are the components of the control system?*

♢ *what does the robot know and keep in its brain?*

♢ .....

The last three problematics, in particular, are treated in highly different ways by each architecture.

We are not going to analyze further every control architecture, since this thesis will mainly focus on the approach used with regard to diverse kinds of programming languages. Although we will, after all, put our attention on the last two mentioned in order to give some consideration about the application of both the approaches concerning the same problem.

1.3 Robot Programming Systems

After the classification of the robot's category and the architecture of the control part of a robot, its "brain", we need to gather those information and according to them, choose a consistent programming system so as we can define a robot software architecture and provide a convenient control. As we have claimed, most of today's robots do not carry out just one simple task furthermore they are more and more exposed to unskilled people, so what turns out is that such autonomous devices must be easier to program and manage than before.

Still better, as the average user will not want to program their own robot at a low level, the programming system we want to implement has to provide the required level of user control over the robot's tasks or rather the right level of *abstraction*. Thanks to that abstraction we can face such complexity, building and adopting suitable framework, architectures and languages.

Robot programming is largely described in literature. Our analysis is grounded on Biggs and MacDonald work[10], where relevant categories of programming systems are showed and thanks to it we can figure out what kind of software architecture implement. Programming systems can be divided first of all into three main categories:

- **Guiding systems**. Where robot is manually moved to each desired position and the joint position recorded.

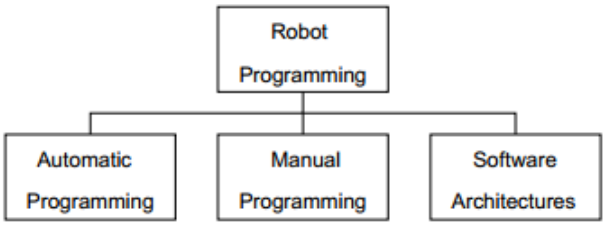

Figure 1.5: Different kinds of Robot Programming.

- **Robot-level systems**. Where a programming language is provided with the robot.
- **Task-level systems**. Where the goals to achieve are specified.

Robot-level systems can be divided again in

- **Automatic programming**. In which system's programmers has little or no direct control over the robot code (like learning systems, programming by demonstration, etc.).
- **Manual programming**. Require the programmer to directly enforce the desired behaviour of the robot, using a graphical or a text-based programming language.
- **Software architectures**. Provide the underlying support as well as access to the robots themselves.

Since the robots we want to program are not actually available, we must create a robot controller by hand and then, this will be loaded into the robot afterwards applying a sort of off-line programming, therefore *manual programming systems* suit good in our context.

In [10] is showed another category subdivision regarding manual programming systems, as we can see in fig. 1.6. Sparing the details that regard another set of subcategories, in our work we are going to use a traditional text-based system with a behavior-based programming language. That is because one of the aims of this thesis is to present an alternative approach to procedural languages that are commonly employed in robot programming. These kinds of languages typically specify how the robot should react to different conditions rather then providing a procedural sequence of actions to apply one after the other.

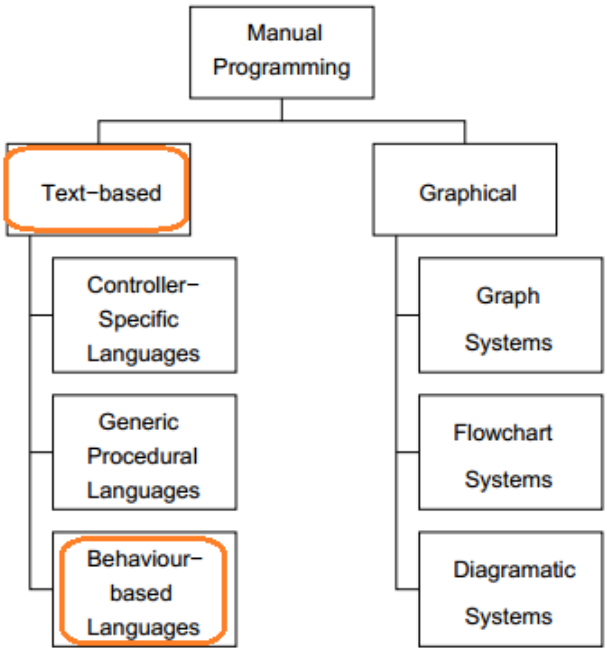

Figure 1.6: Subcategories of manual programming systems.

While controller-specific and generic procedural languages have a big problem regarding the lack of a universal standard between languages from different manufacturers, using a behaviour-based language we can rise to a higher level of abstraction providing a less complex and non-robot-specific way to program the control.

1.4 Benifits and Drawbacks

Over time, robotics and its programming have brought lots of advantages to our society from various points of view. Robots -in particular, the industrial ones- have dramatically improved **product quality** and **speed** of production, with a level of consistency that is hard to achieve in other ways thanks to operations performed with precision and repeatability. They have increased **workplace safety** by moving human workers to supervisory roles so that they are no longer exposed to hazardous circumstances. Additionally, providing high-level platforms and tools to program robots is meaningful to develop smart autonomous devices such as companionship robots for elderly

or blind people assistants.

Nevertheless disadvantages in robotics and programming robots come up. Besides problems regarding the **expense** of the initial investment that companies and people in general have to undertake to purchase robotic equipment either hardware -any kind of sensor or electrical device- or software -development platforms, simulation tools. Morevore people who are pursuing to robotics will require training not just in programming but also for what concern physics, electronics and other fields.

Focussing on those aspects closely related to programming, is not always possible for a programmer to have actually at its disposal all necessary information because of it cannot have all the needed sensors whose in turn could be affected by **noise**. In fact things never go smoothly for robots operating in the real world, in particular, as reported in[12]:

- the robot's program makes an assumption about the world that turns out not to be true,
- a command intended to direct the robot to move in a particular way instead, because of the uncertanty of a real environment, causes the robot to move in a differently,
- the robot's sensor(s) did not react when it should have, reporting a condition that does not exist.

Thus a vital ability we aim to achieve as good robot programmers, is that out robot keep functioning even if things do not go exactly as we expected. Although along with the high complexity of the problems we have to face a considerable range of sensors and related different types of data, for systems whose will act in the real world, what matters is to reach the highest level of autonomy as possible also in unexpected situation even though performance will be affected.

2 Platforms and Languages

In this section we show some of the most relevant platform and tools that everyone interested in robotics it is wise to know in order to make the right

choice according to its own availability in terms of money, time, facilities, expertise and so on. The former subsection will introduce some professional tools whose are at the state of the art in robotics programming, the latter will focus on the role entailed in this field by simulators.

2.1 State of the art

Microsoft Robotics Developer Studio

Also know as MRDS[14], is a Windows based environment for robot control and simulation, aimed at academic, hobbyst and commercial developers that can handles a large variety of robot hardware. It provides a wide range of support to help make it easy to develop robot applications. MRDS includes a programming model that helps make it easy to develop asynchronous, state-driven application by means of what is named *Concurrency and Coordination Runtime, CRR*, a .NET-based library that helps make it easier to handle asynchronous inputs and outputs by eliminating the conventional complexities of manual threading, locks, and semaphore. Another relevant framework is the DSS (*Decentralized Software Services*) which allows to create program modules that can interoperate in order to achieve complex behaviours. MRDS provides moreover, a simple drag-and-drop visual programming language that make easier to program robot applications and a simulation environment (*Visual Simulation Environmet, VSE*) to be able to simulate and test robotic applications using a 3D physics-based simulation tool. This allows developers to create robotics applications without the hardware. Sample simulation models and environments enable you to test your application in a variety of 3D virtual environments.

RobotC

Is probably, the most famous cross-platform robotics programming language (C based) for educational robotics and competitions[13]. It gives programmers a powerful IDE for writing and debugging -thanks to a realtime debugger- robot programs whose can be ported from one robotics platform to another with little or no change in code. The usefulness of such tool is -also- represented by its powerful developing environment that furnishes a great number

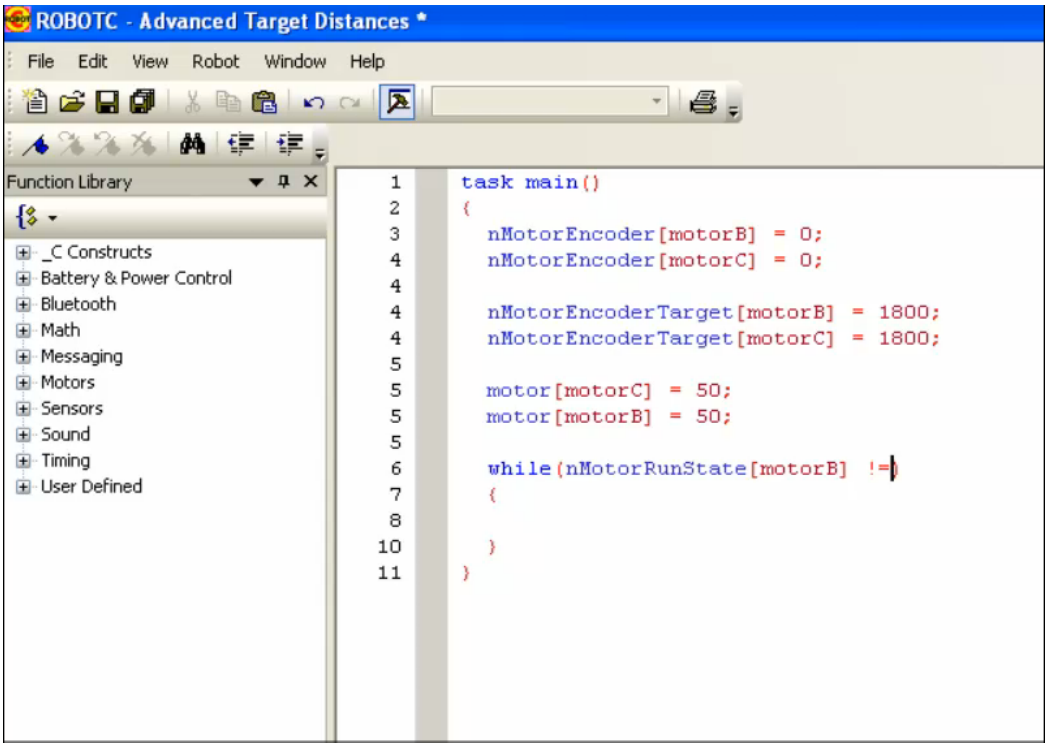

Figure 1.7: RobotC developing environment.

of features such as an user friendly customizable graphic interface, syntax errors detections, drag-and-drop every single variable into the editor, a USB joystick controller integration to drive the robot, and others.

It is currently supported on several different robot hardware platforms like LEGO MINDSTORM, CORTEX, IFI VEX and Arduino as well. Furthermore with **RobotC Real World** it is possible for programmers to test their robots in a simulation environment before they test the code onto a real mechanical system, using the same RobotC code used for simulation in real environments. Maybe the matter that can be experienced by programmers is to use not a standard programming language but a "owner language", RobotC is indeed a language.

ROS

It is an open-source, C++ based, widespread software framework for robot software development which gives operating system functionalities. Those

functionalities are hardware abstraction, low-level device control, implementation of commonly-used features, message-passing between processes, and package management. It is based on a graph architecture where each node receive and process several messages from/to sensor, actuators about their state. ROS -*Robot Operating system*- is composed by an operating system and a suite of package called *ros-pkg* that implement a range of operations like object identification, face recognition, motion tracking, planning, perception, simulation, planning etc. It is released under the terms of the BSD license.

URBI

Like the above framework, URBI is open source and based upon the programming language C++ useful to create robot applications and complex systems. It relies on a **UObject**[4] components architecture and gives a parallel and event-driven script language named **urbiscript** which can be used as a glue among the **UObject** components into a functional behaviour. Thanks to its parallelism and event-driven semantic it turns out suitable for most robot programming and even for complex AI applications.

The goal of Urbi is to help making robots compatible, and simplify the process of writing programs and behaviors for those robots. The range of potential applications of Urbi goes beyond robotics, since it has been successfully used in generic complex systems, where parallel and event-driven orchestration on multiple agents is the rule[16].

2.2 Simulators

With regard to the professional tools and frameworks seen so far, we noticed that the word "simulation" often came out, why? The answer entail some aspects: first of all because sometimes -or rather often, in educational scenarios- the mechanical devices (robots) we want to control are not so cheap, besides, to reach an effective required behaviour we need to apply several number of tests, which concern -for trials on a real robot- a considerable amount of

[4] A C++ component library that comes with robot's standard API.

resources and usually risks like broke components of the robot (and external) or damages to user.

Thus with simulators it is possible di create robot applications without depending physically on the actual machine, saving time and cost. Simulation are stricly connected with **off-line programming**: it takes place on a computer and models of the workcell[5] with robot, pieces and surroundings are used. The robot programs are verified in simulation and any errors are corrected. The biggest advantage of off-line programming is that it does not occupy production equipment, and in this manner production can continue during the programming process. Advanced off-line programming tools contain facilities for debugging and these assist effective programming.

The use of a fast prototyping and simulation software is really useful for the development of most advanced robotics project. It actually allows the designers to visualize rapidly their ideas, to check whether they meet the requirements of the application, to develop the intelligent control of the robots, and eventually, to transfer the simulation results into a real robot. Summing up the main features of programming robots through simulation are:

- Fast robot prototyping
- Physics engines to reproduce realistic movements
- Realistic 3D rendering, used to build the environment in which the robot is situated and interacts
- Testing a certain software or ideas in general onto an autonomous device, saving money and time
- Dynamic robot bodies with scripting (huge range of programming language supported)

However simulation still have few problems: even if we spend a lot of time to make a perfect simulation it is likely impossible to achive a total realistic setting because of the infinite number of issues we have to take into account for reproducing a world with its actual natural and physical laws. Furthermore simulations are often wrong: they are wrong because the experimenter

[5] A workcell is an arrangement of resources in a manufacturing environment to improve the quality, speed and cost of the process. Workcells are designed to improve these by improving process flow and eliminating waste[17].

makes mistakes, or is not sure what features are most important and hence oversimplifies - common for new experimental theories. Often complementing the simulation with real life experiments is meanigful for comparison to make sure the simulation is accurate.

In conclusion, simulation should be used as a complimentary tool, but is not an end-all solution.

3 Webots

Seeking among a range of significant software, one of them results to be well-suited for our aims, **Webots**™. It is worth it to dedicate to Webots its own section, since it is the powerful tool we are going to use to perform following works.

Webots is a professional robot simulator for fast prototyping and mobile robot simulation, widely for educational purposes and online contests. Its developing starts from 1996 by Dr. Olivier Michel at the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland. Since it became a commercial product, in 1998, has been sold to over 400 universities and research centers[6] all over the world[15]. Its main fields of application are:

- Fast prototyping of wheeled and legged robots
- Swarm intelligence (Multi-robot simulations)
- Artificial life and evolutionary robotics
- Simulation of adaptive behaviour
- Self-Reconfiguring Modular Robotics
- Teaching and robot programming contests

As shown in fig.1.8, Webots depicts a robotics project as a four steps activity.

The first stage concerns designing the physical model of the environment by filling it with any kind of object like obstacles, walls, stairs ect. All the

[6] Besides universities, Webots is also used by companies like Toyota, Honda, Sony, Panasonic, Pioneer, NTT, Samsung, NASA, Stanford Research Institute, Tanner research, BAE systems, Vorverk, etc.

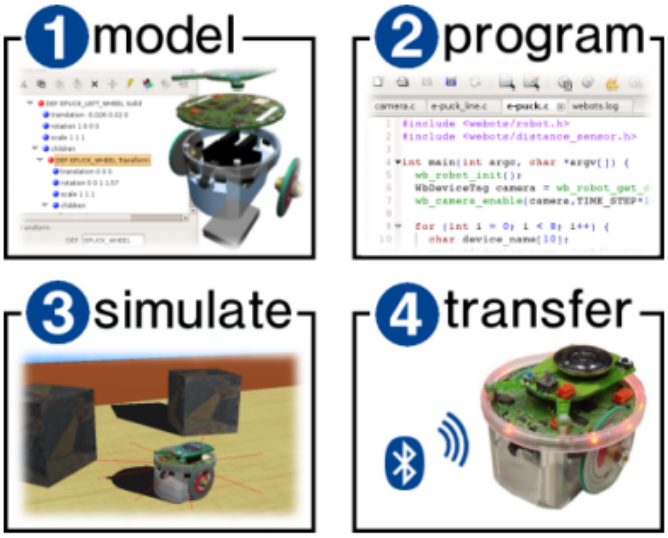

Figure 1.8: Stages that compose a robotics project development using Webots.

physical parameter of these objects such as their mass distribution, friction factor, bounding objects, damping constants can be properly defined in order to simulate their physics too. Afterwards we must make up the body of the robot[7] including limbs, joints, rotor etc. Roughly speaking its set of sensors and actuators. All these components are the building blocks of our robot and we are allowed to modify them (as we like) dynamically in terms of their shape, color, position, physical and technical properties -in case of sensors or actuators.

The second stage consists of robot's behaviour programming. Webots gives a significant range of programming languages that can be used to program the control, like Java, C, C++, Matlab, Python, URBI and allows to interface with third party software through TCP/IP. Usually the robot's control to program runs endlessly gathering sensory inputs, reasoning about these information -the actual core of robot behaviour- to get following action(s) and then send actuators commands to perform them.

The third stage allows the programmer to start the simulation so as to verify whether the robot behaves as expected. In this step we are be able to

[7]From now on we are going to use the singular term, but we are allowed to define and add into the environment as many robots as we want.

see the robot's program in execution and interact dynamically with the environment by moving objects and even the robot. Simulate complex robotics devices -including articulated mechanical parts- requires an exact physics simulation; to achieve this Webots relies on a powerful tool named **ODE** (Open Dynamics Engine), a physics engine used for simulating the dynamic interactions between bodies in space. Webots simulation engine uses virtual time, thus, it results possible to run the simulation much faster than a real robot (up to 300 times faster) An important feature provided is the chance to trigger the *step-by-step* mode to analyse, in detail, the behaviour of the simulation.

The last stage is the transfer the robot's control program into a real robot that will run within a real environment. If we defined the behaviour correctly -in terms of a well suited robot controller and sensors/actuators setting as well as components inside the environment- the real robot should shows more or less the same behaviour of its simulation counterpart. In case this would not happen we have better to go back to previous stages and ensure that we have not make coarse mistakes. If we did not any slip maybe there is the need to refine the model of the robot.

With regards to sensors and actuators, Webots gives complete libraries so that the programmer can exploit their values and send commands. A large choice of sensors can be plugged into a robot: distance sensors, GPS, cameras, light sensors, touch sensors, gyroscopes, digital compasses and so on. Similarly, a handful of actuators can be added as well, like: servo motors (arms, legs, wheel etc.), LEDs, emitters, grippers, pens, displays, linear motors (pistons). Another relevant advantage of using Webots is that we do not need to create our own robot -and its environment- from the scratch every time. Indeed, a lot of world examples and commercially real robot models, like **Aibo**, **e-puck**, **e-puck**, **Lego Mindstorm** and **Khepera** are ready-to-use.

In fig.1.9 is showed the user interface of Webots developing environment. It is composed by:

- **Scene tree** on the left, where a programmer adds new objects and edits all properties of these in the simulated world
- **3D window** in the middle, where is possible view the simulated envi-

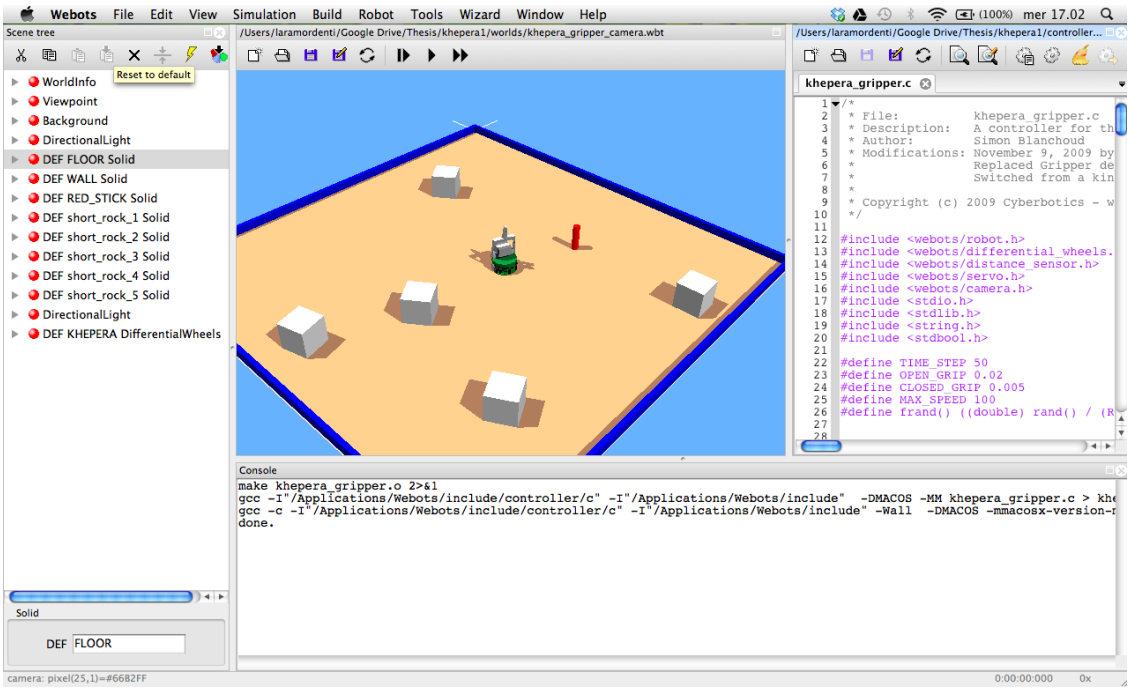

Figure 1.9: Webots user interface.

ronment and interact with it

- **Text editor** on the right, where a programmer write the control code
- **Console** on the bottom, where there are showed build and run time compilation errors/warnings. It plays the role of standard output.

4 Recapitulation

Summing up, the increasing availability of autonomous devices and systems has brought to the need of programming technologies and tools -in robotics area- more affordable and simpler to be used, even by non-robot programmers. However this spreading does not mean that each system or component can be actually at programmer disposal, thus simulators have become probably the most significant tool for those people whose work in robotics field. Surely in this chapter we have not analized deeply every aspects concerning this branch of technology, because of we do not want to overwhelm this thesis by unnecessary details (regarding our purposes). We wished to present just

some key aspects, so as to show off -in the next chapter- how we can program robot's controllers in a new, meaningful way.

Chapter 2

Agent Oriented Programming & BDI-based Programming Languages

After discussing the basics of robotics and its programming, now we are going to talk about the possibility to achieve a meaningful robot programming by means of an *agent-oriented* approach. This chapter provides a brief overview of agent oriented programming (AOP) and agent programming languages (APL), focussing the BDI architecture and pointing out its importance in robotics with regards to BDI agent model. Here are showed those aspects that will turn out useful for the following chapters.

1 Agent Oriented Programming

Firs, we start by considering the typical way to program robots: at least until last years , robot's controllers were programmed through *Functional Programs*. They are so called because, they could be depicted as a function $f{:}I \mapsto O$ from some domain I of possible inputs -the sensor values- to some range of possible outputs -actuator commands. Even thoug this there are a wide range of well-known techinques to develop that kind of programs, unfortunately many programs do not have this simple input-output operational

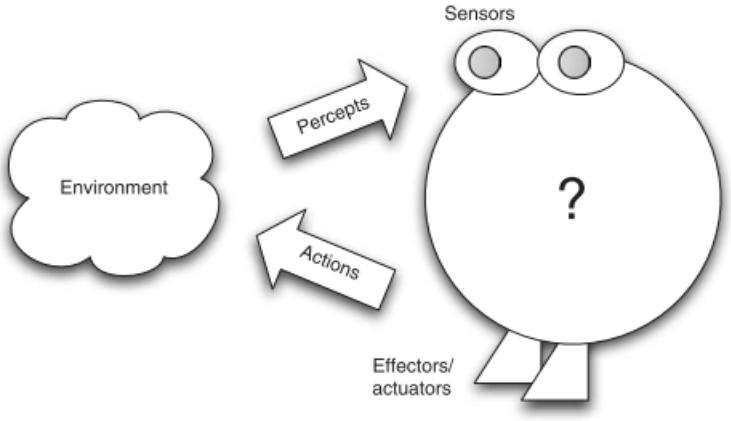

Figure 2.1: High-level interaction between an agent and its environment.

structure. More specifically many of this systems -in particular robotics ones- need to have a "reactive", "time-extended" flavour, in the sense they have to mantain a *long term, ongoing interaction* with the environment. These programs do not want to perform a mere application of a function to an input and then terminate. Thanks to the significant amount of literature that concerns this topic, we know that **agents**, are a relevant (sub)class of reactive systems that turns out to be well-suited for programming robot applications.

The term *agent-oriented* programming was conied in 1989 by Yoav Shoham in order to describe a new programming paradigm based on cognitive and societal view of computation. It was inspired by previous research in AI, distributed/concurrent/parallel programming.

An agent is a system that are *situated* inside an *environment*, that means it is able to *sense* the environment (via sensors) and the ability to perform *actions* (via actuators) so as to modify such environment. The main issue that an agent has to face is **how to decide what to do**[1]. Since the definition of agent is anything but straightforward, we would rather define an agent in terms of its key properties. In [24] Wooldridge and Jennings argued that agents should have the following properties:

- **Autonomy** Typical functional programs doesn not take the initiative

[1]Like we defined in the previous chapter, this is very similar to the question that a robot controller has to answer. This is because it results clear to choose an agent approach to robot programming.

in any sense, they just respond to our inputs. Roughly speaking our aim is to delegate goals to agents, which decide how best to act in order to achieve these goals. Agents are autonomous as they *encapsulate control* and have no interface, so that they cannot be controlled or invoked. *An autonomous agent makes independent decisions about how to achivede delegated goals* without being driven by others.

- **Proactiveness** Agents are proactive by definition[2]: proactiveness means "make something happen" rather than waiting for something to happen. Java objects, for example, cannot be thought as agents, as they are essentially passive (we need to call a method to interact with them).
- **Reactivity** Robotics domains are characterised by highly dynamic conditions: situations change, information is incomplete, resources are scarce, the actions performed are not deterministic in their effects. This means that an agent must be responsive to changes in the environment. However implementing a system that achieves a balance between goal-driven and reactive behaviour turns out to be tough.
- **Social Ability** Represents the ability of agents to *cooperate* and *coordinate* activities with other agents, so as to ensure that delegated goals will be reached. In many applications, have more agents that fulfill a specific part of the overall computation could be useful to achieve a goog level of work balancing.

In fig.2.2 we have depicted the most important categories of software agents, with regard to their features and properties[26]. Briefly: *(i) collaborative agents* are designed to cooperate with other agents, they have the ability to decline an incoming request, *(ii) interface agents* mediate the communication with the user, playing the role of service provider, they are autonomous and able to learn by experience however they have no interaction with ohter agents, *(iii) intelligent agents* autonomous, endowed with the ability to learn and cooperate. This type of agent has got an internal symbolic representation of the surrounding world that helps to choose the right action to perform, so as to fulfill its goals.

[2]The etimology of the word agent is from the Latin *anges* that means "the one who acts".

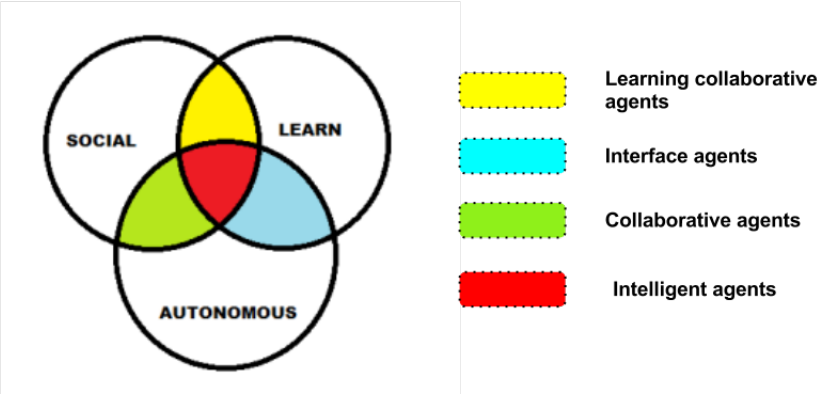

Figure 2.2: Different kinds of software agents, according to their skills.

In practice, systems composed of a single agent are rare: the more common case is for agents that run an environment which contains other agents, called *multi-agent systems*. In such environments, agents communicate with each ohter and control parts of their environment in order to achieve either social or individual goals. Thereby AOP turns out to be suitable to be applied in robotics thanks to the above properties. Then, we can introduce hereafter maybe the best agent model to convey the concept of intelligent entity into a software component.

2 BDI Architecture

As computer systems become ever more complex, we need more powerful abstractions and metaphors to explain their operation. Because of complexity growing, mechanicistic / low level explanations become impractical. Therefore an agent, in order to cope with this increase of complexity, need to have *mental components* such as: belief, desire, intention, knowledge etc, a stateful agent or rather an **intelligent agent** (see fig.2.2). The idea is to use the **intentional stance**[3] as an abstraction in computing in order explain,

[3]When explaining human activity, it is often useful to make statements about whatever is argued to be true or not. These statements can be read in terms of folk psychology, by which human behaviour can be explained and can be predicted through the attribution of mental attitudes, such as believing and wanting, hoping, fearing, and so on[1].

understand, drive the behaviour and then program computer systems.

Agents are explained in terms of mental attitudes, or mental states, whose contain an explicit, *symbolic* model of the world. Every agent makes decisions on what is the next action to perform to reach a desired goal, via *symbolic pratical reasoning*[2] -theory developed by the philosopher Michael Bratman- that could be defined as:

"The activity to choose the action to perform once the next internal mental state is defined, according to the perception of the external environment and its previous internal mental state."

An agent with mental state represents its knowledge with *percepts*, *beliefs* while its objectives are represented with *goals*, *desires*; here arise the concept of BDI (Belief Desire Intention) model. The *intentional system* just explained is used to refer to a system whose behaviour can be predicted and explained in terms of *attitudes* such as belief, desire and intention. The idea is to that we can talk about computer programs as if they have a mental state, thus when we talk about BDI systems, we are talking about computer programs with computational analogues of beliefs, desires and intentions.

Hereafter we are report a roughly definition of each basic element of the model[3][4]

- *Beliefs* are information the agent has about the world, that could be out of date and/or inaccurate. They are expected to change in the future as well as the environment changes. Typically ground sets of literals.
- *Desires* are all the possible state of affairs that the agent might like to accomplish. Having a desire does not imply that the agents acts upon it: it is just an *influencer* of the agent's actions.
- *Intentions* are those states of affairs that the agent has decided to apply. They may be goals that are delegated to the agent or may result from considering options. Intentions are emergent properties reified at runtime by selecting a given desire for achieving a given goal.

This idea of programming computer systems in terms of mentalistic notions such as belief, desire and intention is a key component of the BDI model and they are the basic data structures of AOP. As we mentioned, the tricky

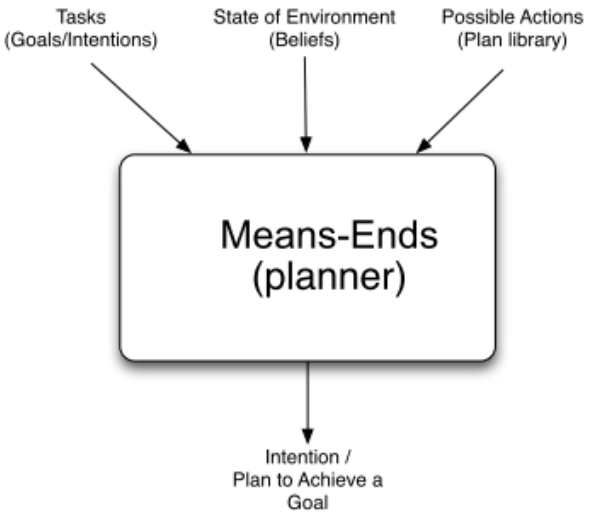

Figure 2.3: Simple representation of inputs and outputs involved in means-end reasoning.

activity is to shift from beliefs, desires and intentions to its actions. The particular model of decision-making underlying the BDI model in known as **pratical reasoning**, defined as:

"Is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes[2]."

It is the **human-based reasoning** direct towards actions, the process of figuring out what to do in order to achieve what desired. It consists in two main activities: *deliberation* and *means-end reasoning*. The former happens when the agent makes decision on **what state of affairs** the agent desire to achieve, the latter happens when the agent makes decisions on **how to achieve** these states.

The output of *deliberation* activity, are the intentions (what agent desires to do/achieve) whereas the output of *means end reasoning* (see fig.2.3) is the selection of a course of action the agent needs to do to achieve the goals. It is widely known -especially in AI- as a *planning* activity that takes as inputs the representations of goals to achieve, the information about the state of the environment and the actions available to the agent, so as to generate *plans*

```
Practical Reasoning Agent Control Loop
1.  B ← B0; I ← I0; /* initialisation */
2.  while true do
3.      get next percept ρ through see(...) function
4.      B ← brf(B, ρ); D ← options(B, I); I ← filter(B, D, I);
5.      π ← plan(B, I, Ac);
6.      while not (empty(π) or succeeded(I, B) or impossible(I, B)) do
7.          α ← head(π);
8.          execute(α);
9.          π ← tail(π);
10.         get next percept ρ though see(...) function
11.         B ← brf(B, ρ);
12.         if reconsider(I, B) then
13.             D ← options(B, I); I ← filter(B, D, I)
14.         if not sound(π, I, B) then
15.             π ← plan(B, I, Ac)
16.     end-while
17. end-while
```

Figure 2.4: Means-end reasoning control loop. B, D, I means respectively, beliefs, desires and intentios.

as course of actions to follow with the purpose of achieving the goals.

Since the *means end reasoning* control loop showed in fig.2.4 in quite far away from the actual implementation, because of we do not know the implementation of each function as well as what is the content of B, D, I. Therefore we are going to introduce another relevant agent architecture: **Procedural Reasoning**[4]. In Procedural Reasoning Systems (PRS), an agent is equipped with a library of pre-compiled plans, manually contructed in advance, by agent programmers -instead of do planning. Such plans are composed by: a goal (post-condition of a plan), a context (the pre-condition of a plan) and a body (the course of action to carry out).

A goal tells us, what a plan is good for, the context part defines what must be true in the environment in order for the related plan, to be true. Finally the body, can be whatever richer than a simply list of sequential actions. As a matter of fact, it is possible to have disjuntions of goals, loops and so forth.

An BDI-based agent comprise three dynamic and global structures repre-

[4]The Procedural Reasoning system, orinigally developed at Stanford research Institute by Michael Georgeff and Amy Lansky, was perhaps the first agent architecture to explicitly embody the BDI paradigm, and has proved to be one of the most durable approaches to developing agents to date. PRS has been re-implemented several times from different universities, so as to create new instances of it.

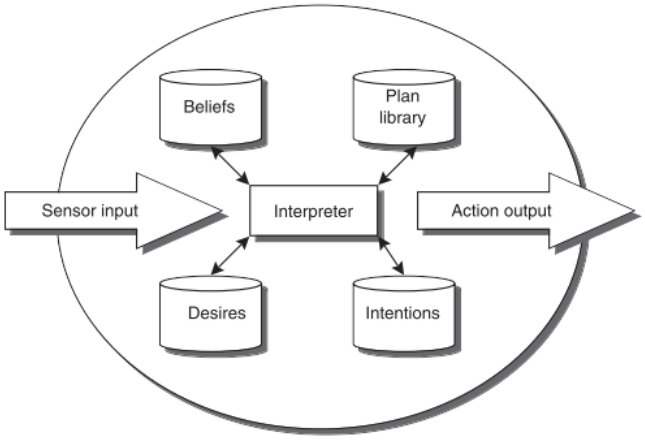

Figure 2.5: Procedural Reasoning System (PRS).

senting its beliefs, desires and intentions, along with an input queue of events. These events could be *external* -coming from the environment- or *internal* -come from some reflexive action.

Summing up, a BDI system are based on sets of:

- beliefs,
- desires (or goals),
- intentions ,
- internal events, in response either to a belief change (updates, deletion, addition) or to goal events (new goal adoption, goal achievement),
- external events, from the interaction with external entities (signals, incoming message, etc.); and
- a plan library

As we have mentioned previously, this model is another important category of control architecture on which robots programming can be based despite of the well-known architectures are much more widespread in that area.

3 Agents & Artifacts meta-model

Robert Amant and Donald Norman, in their articles remarked the fundamental role that *tools* and, more generally, **artifacts** play in human society. They wrote:

"The use of tool is a hallmark of intelligent behaviour. It would be hard to describe modern human life without mentioning tools of one sort or another[6]."

"Artifacts play a critical role in almost all human activities [...] indeed, the development of artifacts, their use, and then the propagation of knowledge and skills of the artifacts to subsequent generation of humans are among the distinctive charactetistics of human being as a specie[9]."

These considerations arose because of almost any cooperative working context accounts for different kind of object, tool, artifact in general that humans adopt, share and properly exploit so as to support their working activities. Such entities turn out to be fundamental in determining the success of the activitiesas well as their failure. According to the aims -not just of this thesis- we think that a robot controller, composed by agents could take several advantages using different sorts of artifacts. In conclusion, we introduce -and then use, further- a programming meta-model called **A& A** (Agents and Artifacts) to model and engineer the working environment for a society of intelligent (cognitive) agents.

Artifacts

Roughly speaking we may define an artifact as *a computational entity aimed at the use by an agent*. Given that we claim that an artifact are designed for use, to serve some purposes, so when designing anm artifact we have to take into account their *function* rather than the actual use of the artifact by the agent.

An artifact has to comply with two basic properties: it should ensure *transparency* and *predictability*. The first property is important because, in order to be used by an agent, artifact function sould be available and understandable by agents, whereas the second is needed to promote agent's use since artifact behaviour should be predictable. Essentially it is designed to serve and be governed, an artifact is not autonomous, is a tool whereby an agent is endowed. Hence it is *totally reactive*.

The functionality[28] of an artifact is structured in terms of **operations**, whose execution can be triggered by an agent through artifact **usege interface** composed of a set of **controls** that agents can use to trigger and control

operation execution. Besides the control aspect, the usage interface can provide a set of **observable properties**: properties whose dynamic values can be observed by agents without -necessarily- interact with the artifact. The executio of some operations upon the artifact could cause a series of **observable events** like observable property changes or signals[5]. Finally, more artifacts can be linked together in order to enable an artifact-artifact interaction as a principle of composition, by means of *link interfaces*. This feature ensures both to define explicitly a principle of composability for artifacts - allowing to achieve a complex artifact by linking together simplier ones- and to realise distributed artifacts by linking artifacts from different workspaces (and different network nodes).

Summing up, artifacts are coarsely subdivided into three categories:

- *Personal* artifacts, designed so as to ensure functionalities for a single agent use
  - agenda, timer, etc.
- *Social* artifacts, designed to provide some kind of global functionalities, concerning coordination, organisation, comunication etc.
  - blackboard, tuple spaces, bounded buffers, pipes, etc.
- *Boundary* artifacts, designed to wrap the interaction with external systems or to represent the external system inside the MAS.
  - GUI, Web Services, etc.

In a system that adopt the **A&A** meta-model, a working environment in conceived as a dynamic set of artifacts -whose support system's working activities- organised in *workspaces*. A workspace is a container of artifacts, useful to define the topology of the environment and provide the notion of locality for agents, in order to move towards a distributed multi-agent system (MAS). In fact, different workspaces could be spread over the network and each agent could run on different internet nodes.

A&A MASs are made of pro-active autonomous agents and reactive artifacts whose provide some functions: the interaction between these kinds

[5]Every artifact is meant to be equipped with a manual, which describe the artifact function(purpose), the usage interface and the operating instructions (like a protocol, or better how to correctly use the artifact so as to take advantage from its functionalities.

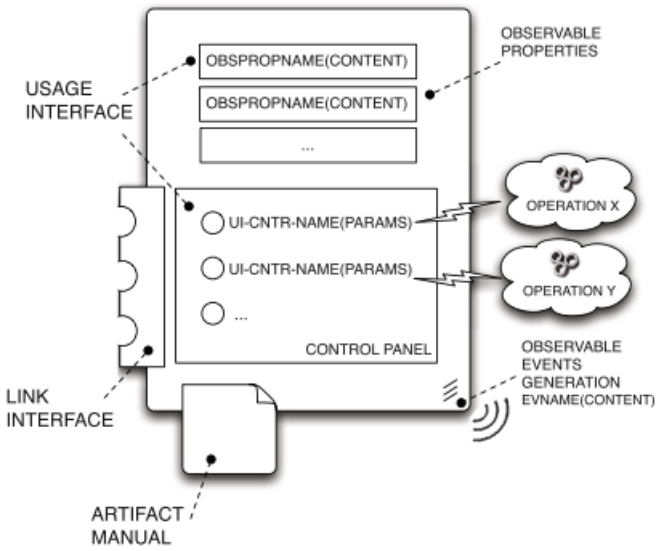

Figure 2.6: Abstract representation of an artifact.

of entities generates the overall behaviour of the MAS. In those systems, such fundamental entities give raise to four different sorts of admissible interactions[5]:

- **communication** agents *speak* with agents
- **operation** agents *use* artifacts
- **composition** artifacts *link* with artifacts
- **presentation** artifacts *manifest* to agents

Finally, from [29] we can see how the overall A&A meta-model could be depicted (see below).

3.1 CArtAgO

Besides the **A&A** abstract programming model, we are going to present the actual, concrete technology which aim is to be used to experiment that model in this thesis: the **CArtAgO** technology.

CArtAgO (**C**ommon **A**rtifact **I**nfrastructure for **A**gent **O**pen environment), is a framework that providing essentially:

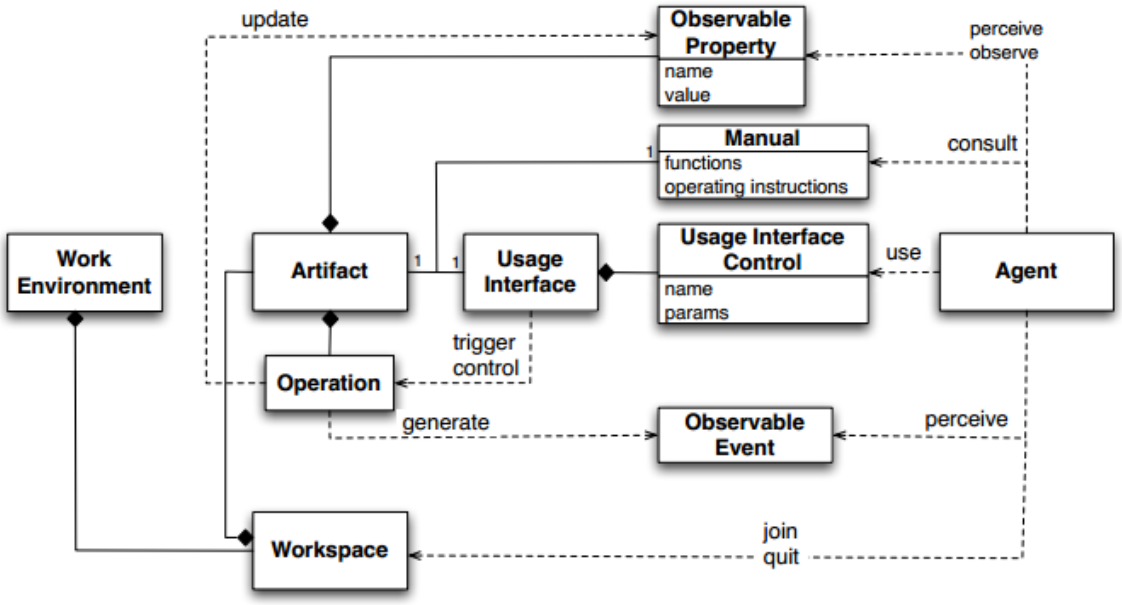

Figure 2.7: A&A meta-model depicted in UML-like notation.

- Suitable API for agents whose work with artifacts and workspaces. By means of proper API that aim at extending the basic set of agent actions with a new one, so as to *create*, *dispose* and *interact* (with) artifacts through their usage interface.
- The capability to define new artifacts type. Thanks to these API, a programmer can implement new types of artifacts by extending the basic class **Artifact**, new operations and operations step by defining methods tagged by **@OPERATION** and **@OPSTEP**. Moreover it is possible to write the artifact function description and the list of observable properties, explicitly declaring the **@ARTIFACT_MANUAL** annotation before the class declaration.
- A runtime dynamic management of working environments. Conceptually it is the virtual machine when artifacts and agents are instantiated and managed: it is responsible of executing operation on artifacts and gathering observable events generated by artifacts.

The CArtAgO architecture implements suitably the abstract model de-

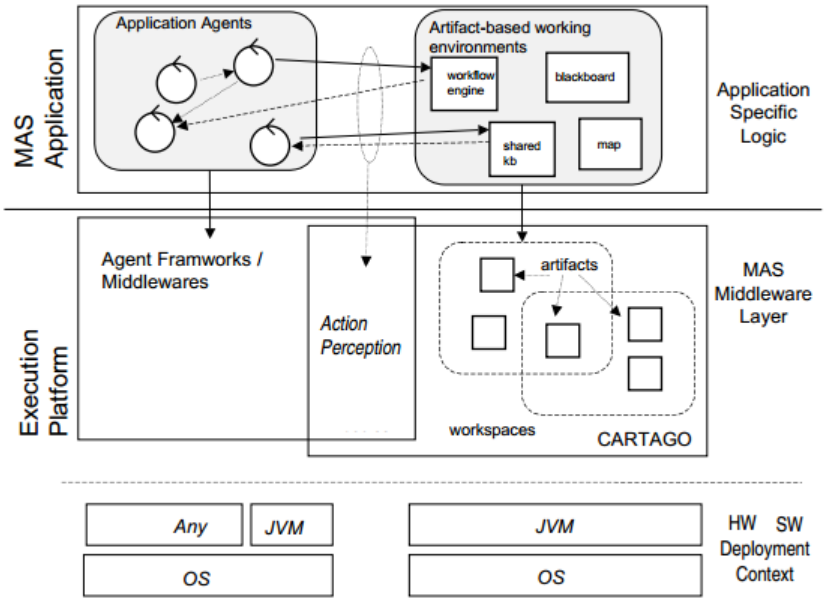

Figure 2.8: MAS exploiting the CArtAgO working architecture.

scribed above, indeed it does not introduce any specific model or architecture for agents and their societies, but it is meant to be integrated and used with existing agents platforms -and languages, as showed further. The **CArtAgO** working environment is composed by three main building blocks as we can see in fig.2.8, *agent bodies*[6] that make the agents situated in the working environment , *artifacts* useful to structure the working environment and *workspaces* as logical containers of artifacts, whose define the topology of the environment.

[6]They are what actually connect the agent *mind* and the working environment. It contains actuators/effectors to perform actions upon the environment and sensors to collect information from it. The concept of "agent" that we conceive so far, is the part that actually govern the agent body in order to perceive the events generated by artifacts -and collected by sensors as stimuli- and execute actions provided by agent mind, so as to affect the environment.

4 APLs (Agent Programming Languages)

The growing studies about MAS have brought to the development of programming languages and tools, that are suitable for the implementation of those systems. Analyzing the literature, several proposals for APL come out: some are implemented from the scratch, others are obtained by extending existing languages that satisfy some required aspects concerning agent programming issues. Using these specific languages rather than more conventional ones, turns out really useful when a problem we have to face is modelled in an agent-oriented fashion (goals to reach, beliefs about the state of the world etc.). In spite of the significant number of languages and tools that has been developed over time the activity regarding the implementation of a MAS still tough because of the lack of specialised debugging tools and required skills that are necessary to map agent design concepts to programming language constructs.

From [18] we know that APLs can be roughly divided in: *purely declarative*, *purely imperative* and *hybrid*. These languages -not all- have some underlying platform which implements the semantics of the APL, but we are going to give just few hints. Hereafter we are going to give a brief overview of most considerable languages for each class and afterwards what language we chose for developing our following explorations.

Hybrid approach

In order to combine significant features of both imperative and declarative languages, an hybrid approach turns out to be a good choice. This programming languages are declarative while at the same time provides some constructs, useful for using code implemented with an imperative programming language -so as it is possible to use legacy code. **3APL**(An Abstract Agent Programming Language) is a language for implementing cognitive agents that have beliefs, goals and plans as mental attitudes. The main task in 3APL consists in programming constructs to implement mental attitudes of an agent as well as the deliberation process thanks to which those attitudes can be manipulated. These specifications are: beliefs, goals, plans, actions (building blocks of plans) and reasoning rules. As we mentioned before, it supports

the integration of Prolog and Java, where the former is declarative and the latter is imperative. Others well-known hybridi programming languages are **Go!** and **IMPACT**[7].

Probably the most widely APL, which it is worth of a dedicated section, since it will be the approach used during the course of this thesis.

4.1 Jason

Jason is an extension of *AgentSpeak*[21][8] programming language, which has been one of the most influential abstract languages based on the BDI architecture. Jason is the first fully-implemented interpreter, it is Java-based and open-source and additionally, if we decide to implement a MAS in Jason, this can be effortlessly distributed over the network.

As Jason is a BDI architecture based language, we already know the basic components whose compose the agent: a *belief base* that are continuously updated according to changes in the perceived environment, the agent's *goal* which are reached by means of the execution of *plans* -present inside the *plan library*- that consists of a set of *actions* whose change the agent's environment in order to achieve its goal(s). These changes in the environment are applied by another component of the architecture, according to the *choices* selected on the course of actions.

The interpretation of the agent program determines the agent's **reasoning cycle**, analogous to the BDI decision loop seen previously in fig.2.4. The agent cycle can be described -sparing the details- as the constant repetition of these ten steps: *(i) perceiving the environment*, *(ii) updating the belief case*, *(iii) receiving communication from other agents*, *(iv) selecting "acceptable" messages*, *(v) selecting an event*, *(vi) retrieving all relevant plans*, *(vii) determinig the applicable plans*, *(viii) selecting one applicable plan*, *(ix) selecting*

[7]To retrieve meaningful information about these hybrid languages, it is possible to look through their official web sites. 3APL `http://www.cs.uu.nl/3apl/` , Go language: `http://golang.org/` , IMPACT: `http://www.cs.umd.edu/projects/impact/`

[8]The AgentSpeak language introduced by Rao, represents an attempt to distill the key feature of the PRS into a simple (fig.2.5), unified programming language. Rao wanted a programming language that provided the key features of PRS, but in a suffciently simple, uniform language framework that it would be possible to investigate it from a theoretical point of view, for instance by giving it a formal semantics.

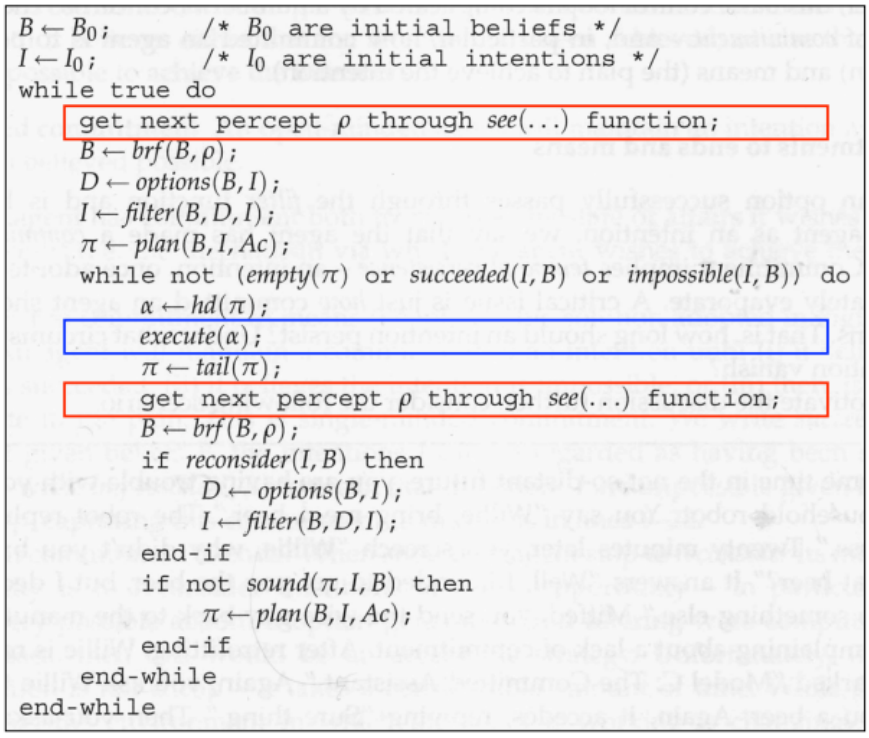

Figure 2.9: Agent's reasoning cycle expresses in pseudo code.

an intention for further execution and *(x) executing one step of an intention.*

Before restarting the cycle, in case there are some some suspended intentions, waiting for a feedback action execution or message reply, the interpreter checks whether the feedback/message are available, and if it so it updates the intention including that in the set of intentions, so that it may be selected in subsequent steps *(ix)*.

Summing up, a Jason agent program is composed of a *belief base*[9] at the beginning,

```
colour(box1,blue)[source(bob)].
~colour(box1,white)[source(john)].
colour(box1,red)[source(percept)].
```

Besides beliefs that the agent has got initially, we can also provide goals that the agent will attempt to achieve from the start, the *achieve goals*, for example:

[9]Within the square brackets are specified the belief **annotation**: complex terms providing details that are strongly associated with that particular belief.

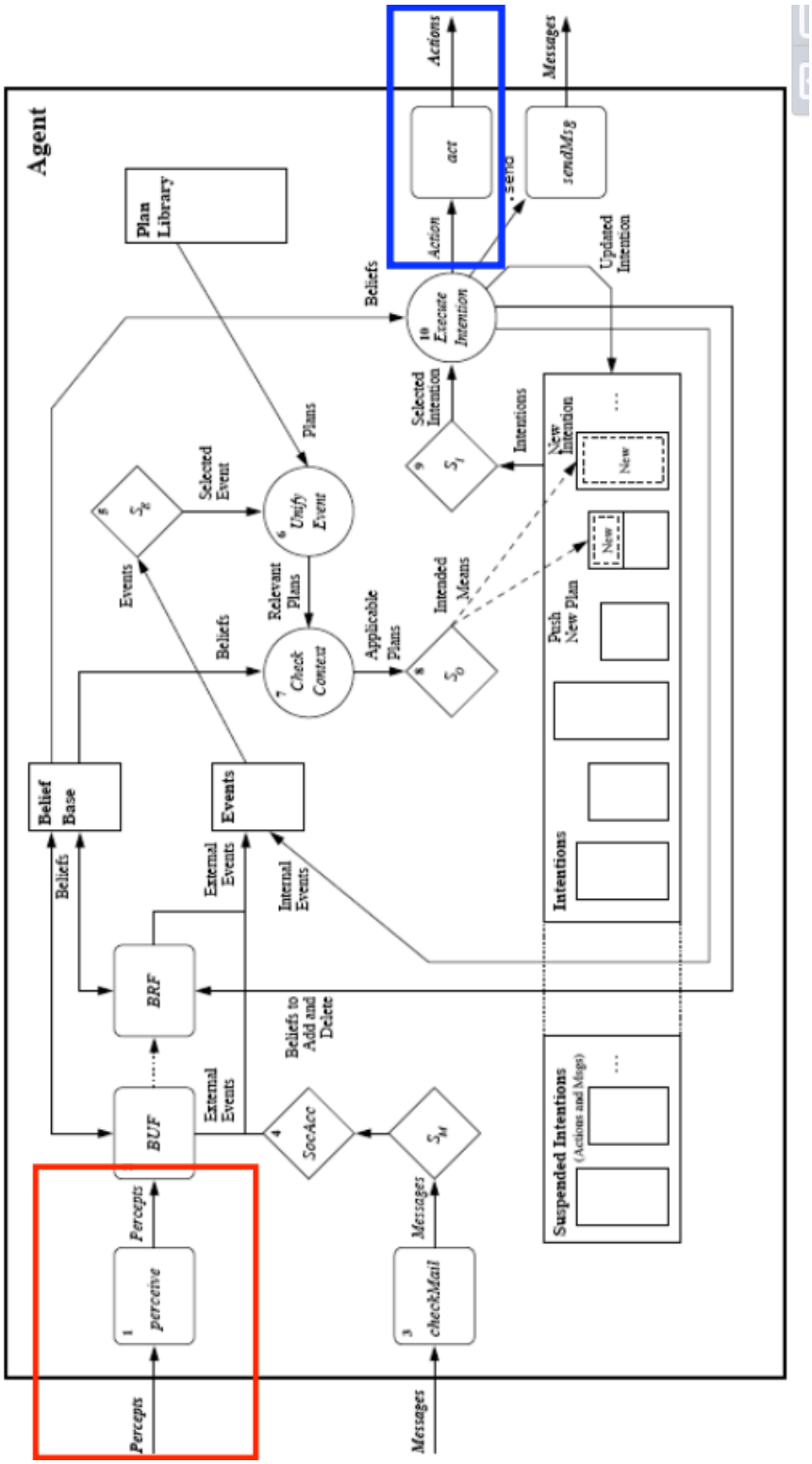

Figure 2.10: Jason reasoning cycle expresses through its architectural counterpart, from [27].

Notation	Name
$+l$	Belief addition
$-l$	Belief deletion
$+!l$	Achievement-goal addition
$-!l$	Achievement-goal deletion
$+?l$	Test-goal addition
$-?l$	Test-goal deletion

Figure 2.11: Types of triggering events.

```
!findBox.
```

Beliefs and goals, are two important mental attitudes we can express in the actual agent source code. The third essential construct of a Jason agent program are *plans*. These are composed by three parts:

```
triggering_event : context <- body.
```

- *triggering event*: tell the agent, which are the specific events (see fig.2.11) for which a certain plan will be used;
- *context*: very important for reactive planning systems in dynamic environments. The context of a plan is used exactly for checking the current situation so as to determine whether a particular plan, among the alternative ones, is likely to succeed in handling the event;
- *body*: is simply a sequence of formulae determining the course of actions. Not all these actions aimed at changing the environment, some of them could start new goals, called in this case *subgoals*.

Therefore a plan can be written as:

```
+!boxFound: colour(_,blue)[source(S)]
<- println("Hi! ",S," I've found your blue box!"); !findBox.
  ...
```

Furthermore between beliefs and initial goals it is possible to add to source code a set of rules, written in Prolog, which allow programmers to conclude new things based on things that are already known. Including such rules in an agent's belief base can simplify certain tasks

```
likely_colour(C,B)
      :- colour(C,B)[source(S)] & (S == self | S == percept).
```

The first rule says that the most likely colour of a box is either that which the agent deduced earlier, or the one it has perceived.

We can now conclude this section, showing few features provided by the Jason language[10], besides interpreting AgentSpeak.

- *Strong negation*, so it is available not only the close-world assumpion, but also the open-world,
- *handling plan failures*,
- *annotations*, so that beliefs can be enriched with meta-level information and can be used by elaborate selection functions,
- possibility to run a multi-agent system distributed over a network using JADE or Saci, or other user-defined distribution infrastructures,
- fully customisable selection function and overall agent architecture (in Java)
- support for developing environments -not programmed in AgentSpeak but in Java,
- straightforward extensibility by user-defined internal actions,
- an *IDE* both in the form of jEdit and Eclipse plugin, along with a "mind inspector" that plays the role of debugger.

4.2 JaCa

An application in **JaCa**[20] is designed and programmed as a set of agent which work and cooperate inside a common environment. Programming the application means then programming the agent on the one side -encapsulating the logic of control of the task to perform- and programming the environment on the other side -as the abstraction that providing the actions and the functionalities exploited by the agents to fulfil their tasks. In **JaCa**, Jason is adopted as a programming language to implement and execute agents and CArtAgO -that, in turn, follows the **A&A** meta-model- as the framework to program and execute the environment where agents are situated. **JaCa** programming model integrates Jason and CArtAgO so as to make the use of artifact-based environment by Jason agents seamless. As a result,

[10]Retrieved official website: `http://jason.sourceforge.net/wp/description`

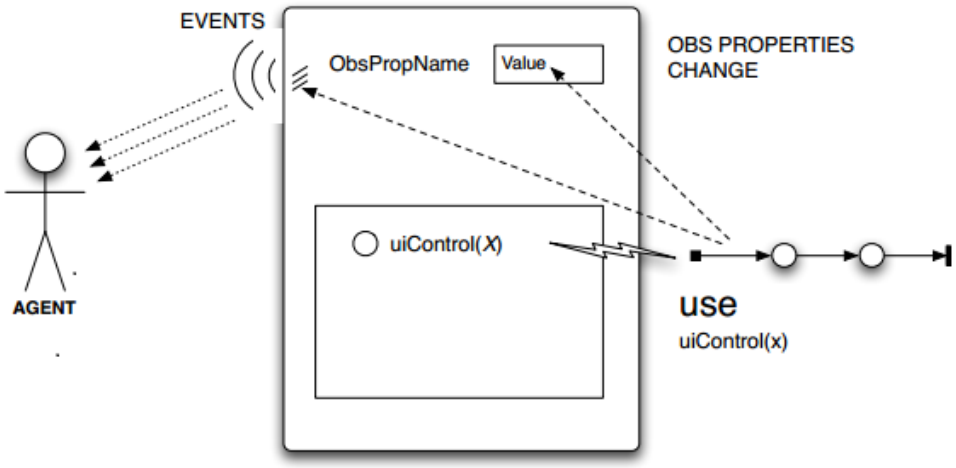

Figure 2.12: Interaction between a Jason agent and a CArtAgO artifact.

the overall set of external actions that an agent can perform, is determined by the overall set of artifacts available inside the workspace in which such agents are situated. Moreover, the whole set of percepts that Jason agents can observe is given by the observable properties and observable events of the artifacts available in the workspace at runtime. If a Jason agent want to exploits an artifact, sensing its events, observing its properties and using its functionalities, it has to explicitly declare its interest by means of a specific action named *focus*. Once we call that action, artifacts' observable properties are automatically added -as beliefs- to agent's belief base and every time such properties change, connected beliefs are updated accordingly. Interaction through *signals* are slightly different: indeed a signal is not added to the belief base but is processed as a non persistant percept possibly triggering a plan.

So a Jason agent should specify plans that react to changes of beliefs that concern observable properties -or trigger a relevant plan according to the value of that belief- and plans that react to incoming signals -that may come not only from artifacts, like in the case of message receipt events. In fig.2.12 we can depict the interaction among Jason agents and CArtAgO artifact in terms of events and utilization.

We think it is worth showing a simple example of the interaction between a Jason agent and CArtAgO platform so as to be not overwhelmed by theoretical details. This example is retrieved from CArtAgO official documentations[19] and helps us to understand better **JaCa** execution, even though it is a classic -trivial- *counter* example. The artifact concerned plays the role of a simple counter with an internal value, providing the increment operation. Such artifact is defined like this:

```
public class Counter extends Artifact {
  void init(){
    defineObsProperty("count",0);
  }

  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("count");
    prop.updateValue(prop.intValue()+1);
    signal("tick");
  }
}
```

The init method represents the artifact constructor that will be executed when the makeArtifact action is performed. In this method the artifact defines, by means of the defineObsProperty primitive, all its properties whose will be observable by agents. In this case the observable property is the tuple "count" with one numerical argument -of integer type- which starting value is 0. The operation provided by the artifact is that method annotated with @OPERATION, called inc, which take no parameters and is meant to be used in order to increment the value of the internal "count" variable. An artifact provides the getObsProperty primitive to retrieve the property and then the updateValue to change the value of it. When the internal value is updated, then a signal is generated so as to notify the agent.

The agent in turn is implemented as we show below:

```
!observe.

+!observe : true <- ?myTool(C); // discover the tool
                    focus(C).

+count(V) <- println("observed new value: ",V).
```

```
+tick [artifact_name(Id,"c0")] <- println("perceived a tick").

+?myTool(CounterId): true <- lookupArtifact("c0",CounterId).

-?myTool(CounterId): true <- .wait(10);
                             ?myTool(CounterId).
```

Through the lookUpArtifact primitive provided by the CArtAgO environment, the agent can discover the identifier of the artifact with the specified name (in this case "c0"), so that it can put the focus on it (thanks to the primitive focus(ArtifactId). By putting the focus onto an artifact, observable properties are mapped onto agent's belief base. So, changes to the observable are detected as changes inside the belief base (in this case +count(V) would be the triggered event.). Furthermore by focussing an artifact, signals generated by an artifact are detected as changes in the belief base, in this example +tick is the event.

The last step regarding the use of both Jason and CArtAgO, concerns the definition of the environment in which the agent runs. Indeed we have to specify in the MAS settings,

```
MAS counter_example {
     environment:
        c4jason.CartagoEnvironment

     agents:
        counter_user agentArchClass c4jason.CAgentArch;

    /*classpath definition*/
}
```

That is, the agent can exploit the CArtAgO APIs.

5 BDI languages for Robot Programming

So far we have pointed out how relevant is the choice of an BDI programming language, so as to provide a better support for implementing autonomous robotic control systems. However someone might wondering: *is this the only exploration of the BDI approach in robot programming?* or

is this approach already explored and tested?.

The answer is, yes.

In fact, the aim of the thesis regarding the exploration of Jason -or better, **JaCa** - as the language to program robot's control and thus using the BDI control architecture, which are anything but unknown in this field. Many other projects have chosen a BDI approach to create robot control programs. One of these are clearly presented in the paper [23], that summarizes a research by Luxembourg and Utrecht universities with the aim to provide necessary methodologies and requirements to facilitate the use of BDI-based APLs for implementing robotic control in a modular and systematic way.

In that project the working team used -for an application scenario- a NAO robot[11] the 2APL BDI programming language along with a robotic framework like ROS (already mentioned both, during this paper). The integration of both of them is a fundamental requirement to facilitate controlling and communication with functional modules developed in this framework (e.g face/voice recognition and a number of high-level actions such as sit-down, turn-neck etc.). This can encourage the use of APLs by robot community and facilitate their use for rapid prototyping and development of autonomous systems.

Another requirement was to provide deliberation capability and, at the same time preserving reactivity. To ensure this requirement the researchers thought to adopt one of the most well-known hybrid architectures, like the classic three layered architecture.

Moreover, the researchers focused on different issues whose have to be faced for developing such sort of systems, such as *real-time reactivity*, *raw data processing* and *monitoring and management of parallel execution of plans.*

[11]NAO is a programmable humanoid robot sold by Aldebaran Robotics and adopted by many universities for academic purposes. See `http://www.aldebaran-robotics.com/en/Discover-NAO/Key-Features/hardware-platform.html`

6 Recapitulation

In this chapter we have seen the basic abstractions about agent oriented programming such as agents, artifacts, architectures, explaining the most significant concepts. These have been brought to practice with the **JaCa** programming technology, which we are going to use in order to perform some explorations in the next chapters. Such choice is due to **JaCa** characteristics whose fit suitably with the requirements of robots controllers. As a matter of fact a robot could be depicted as an autonomous entity that sense and proactively interact with the surrounding environment by means of passive devices like sensors and actuators. These devices represent fundamental tools for the robot's lifecycle and can be considered -from our point of view- as the *boundary artifacts*. Given those reasons we figured that using an BDI-based agent oriented programming architecture could become a right way to program robot's "brain", probably enhancing the current and more conventional programming methods. So in the next chapter we will show how to apply our approach in programming robots thanks to the Webots simulator.

Chapter 3

Using the BDI architecture for Robot Programming: A Jason-based Approach

As for developing this thesis, we will not have a real mechanical system to be programmed. Therefore, as discussed in Chapter1 we need to perform a significant number of simulations of that by means of a simulator (Webots). Moreover, in Chapter2 we have seen the set of approaches to robot programming, focusing on the BDI model and explaining why such choice is relevant in robotics area. In this chapter we are going to talk about the issues brought by the Jason APL from the point of view of a robot programmer. Afterwards we are going to figure out how we could design the agent system, using the **JaCa** programming architecture. So then how we discuss how to integrate the agent system with the simulator so as to depict the overall system we will use in the course of this thesis.

1 Jason for robot programming

Starting from the conclusions of the previous chapter, we can assert that the controller of a robot is going to be mapped into a suitable agent. Indeed, a robot is a physical agent, it has both a computational and physical nature

-complexity of physical world enters the agent boundaries, and cannot be confined within the environment. Besides, robot is intrisically situated, because its intelligent behaviour cannot be considered as such separately from the environment where the robot lives and acts.

Now, the question is:

How many agents shall we program in order to define a robot controller?

This is an important design issue that depends on many aspects. On the one hand, a multi-agent system avoids work overload and allows to allocate a specific task to a specific agent, on the other hand, if the problem must not achieve very complex goal or the subtasks whose compose the main activity are not so tricky we should use just a single agent. In fact in the latter case, if we decide to use more than one agent we have to tackle -as a consequence- others issues like coordination and communication between agents activities. These issues involve a rise of complexity and then the benefits brought by the multi-agent approach may be nullified. With regard to further explorations that entail a single robot which perform tasks that are not too tough, we are going to develop a single agent, representing the robot "brain".

1.1 Layered Architecture

This is a fundamental stage that we have to go through so as to start with the experiments, we start to reason about how we ought to organize the control program. We refer to control architectures we have seen in Chapter1, sorting out the program code according to different approaches for example some could be purely procedural like C and its libraries, or RobotC that has its own architectural elements inside the code (e.g RobotC *behaviour*).

Using agent programs to create **cognitive layer** in a robot control architecture is natural and provides several benefits. Similar to the others approaches, a BDI-based robotic control system also need a **functional layer** to interface with robotic hardware and provide sensory inputs and actions at different levels of abstraction.

Designing and developing software control architectures for robots poses several challenges. Robots are embedded systems that operate in physical,

dynamic environments and need to be capable of operating real-time, in addition a range of perceptions and motor control activities need to be integrated in the architecture. So, providing a proper balance between *deliberation* and *reaction* has been always a major concern in research on robotic control systems. Indeed, deliberation capability is desired for autonomous robots, however it requires on the other hand, a time-bounded reactivity to events it receives from the environment, with the purpose of ensuring robot's safety. Moreover, for the system can be particularly difficult to generate meaningful symbols to represent sensor data or to perform some low-level tasks such as control of motors. These kinds of activities are delegated to components in other lower layers.

Summing up, to address the above issue we believe that is necessary to structure the overall system using a classic **three-layers architecture**, in a way that the highest level has got neat and straightforward information -that are provided by the lower layers.

- The main functions for controlling a robot and supervising the temporal execution of the plans are placed in the *(symbolic) cognitive layer*.

- The *functional layer* is interfaced with hardware and provides low-level perception and action capabilities to higher layers. It is also responsible for interpretating of symbolic messages that represent actions and making robot perform these actions in its physical world.

- The *executive layer* residing between the other two and its main functionality -besides manage the interaction between them- is to refine plans into low-level actions. Furthermore it provides mechanisms to overcome the issue that sensory data is typically noisy, incomplete, quantitative measurements, whereas the cognitive layer needs the symbolic representation of those information, to support logical reasoning[25]. So, this layer process raw sensory data into sensory information at **different level of abstraction** to be used by the control component of the robot for the actions decision process.

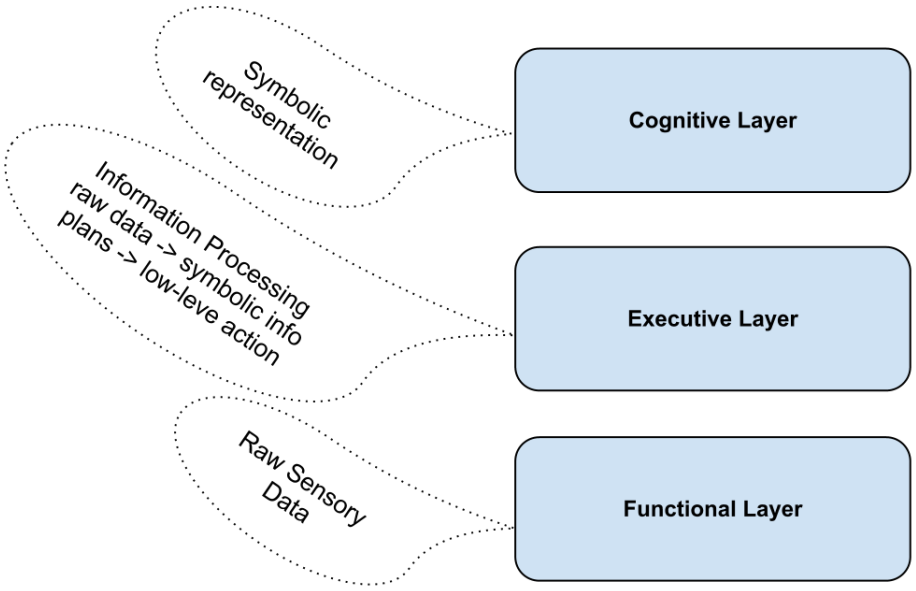

Figure 3.1: First sketch of our three-layer software system.

2 System Overview

Identifying the layers that allow us to separate properly each concern turns out significant, in order to focus on those aspects regarding the strategy to solve the encountered problems. The next stage we must go through is to identify who are the **JaCa** (and Webots) counterparties that can be mapped onto each layer roughly depicted in fig.3.1.

Obviously, in the highest level will reside the cognitive intelligent agent, written in Jason. Althought by choosing the BDI architecture we must take into account another issue

How can we model the perceptions?

In the BDI model, all the robot's perceptions are mapped into proper beliefs inside the knowledge base of the agent, so that it can be constantly aware about the state of the world and act accordingly. Besides, since we are going to use **JaCa** and then the Jason programming language as well as the CArtAgO technology, we know that agents exploit one or more artifact (their functionalities) to fulfill their tasks. So, in order to structure the software system as simple as possible, the executive layer is composed by a suitable

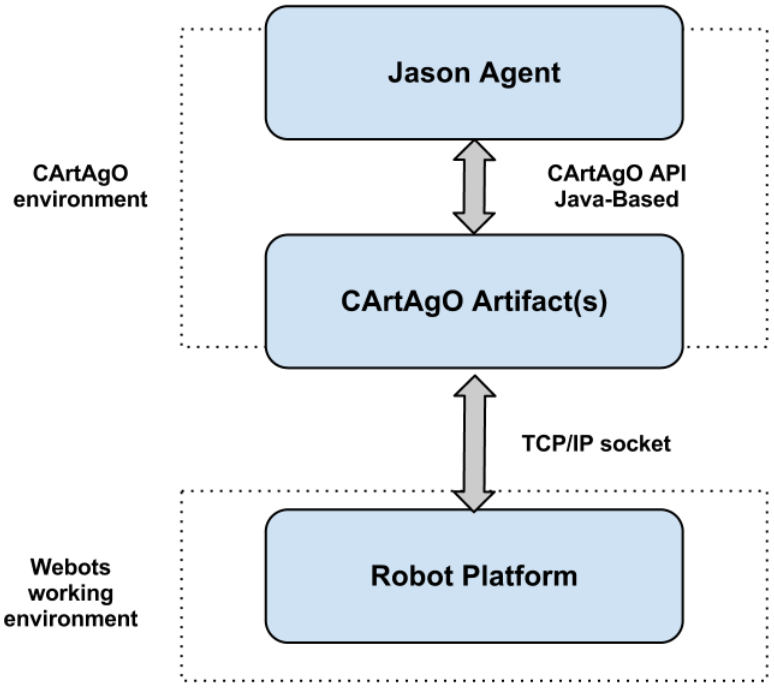

Figure 3.2: Another coarse representation of the system, pointing out the interactions among the layers.

CArtAgO artifact which perform the activities mentioned above. Finally, the lowest level will be represented by the robot platform provided by Webots simulator (this aspect will be analyzed deeply, later).

2.1 Interaction

Each layer has to interact with its adjacent one. The kind of such interaction is very important because these affect the overall behaviour and execution of the system.

The agent and the artifact -cognitive and executive layers- communicate by means of useful Java-based API provided by the CArtAgO technology. To make this possible, we -as programmers- have to define that the controller agent working inside the CArtAgO environment (see 2.3). Whereas between

the artifact and the robot platform -representing the functional layer- we need to employ a basic mechanism, like TCP sockets, since one side will run upon an agent environment while the other side will work inside its own environment (the simulator).

The lack of well-designed interactions bring inevitably to a wrong / unfeasible / non-fluid execution of the system.

2.2 Implementation

As we said in the chapter about agent programming languages, some of them are endowed with IDEs so as to make easier and fruitfully the program development. Thus, we are going to exploit the Eclipse IDE along with its Jason plugin and the CArtAgO platform library[1]. The implementation of that system has been possible thanks to few simple steps.

- Initially we have created a new Jason project from the wizard menu,
- once the project is created we can insert agents and java file (artifacts and extensions of agent class) and implement their properties and interactions,
- then we can set the whole system such as, the *infrastucture*, the *environment*, the source path where retrieve the agent files (.asl) and the list -and the number- of them (see fig.3.3).

Concerning the use of the artifact from the agent, this last point is fundamental, because here we must define that the agents of the MAS are situated inside a CArtAgO environment, so as to exploit its API and thus, use artifacts' functionalities. To express this, we specify:

```
environment: c4jason.CartagoEnvironment
```

In conclusion, the last step to execute is the integration of agent environment with the simulator.

[1]See official websites. Eclipse Juno `http://www.eclipse.org/juno/`, Jason Eclipse Plugin Install `http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/` and CArtAgO on SourceForge `http://cartago.sourceforge.net/`.

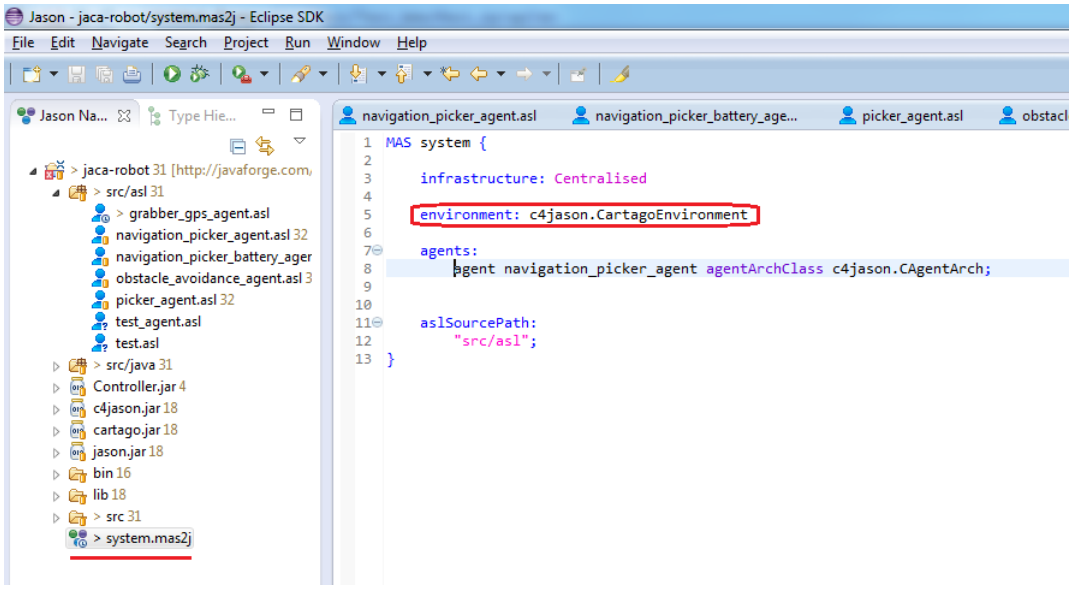

Figure 3.3: MAS setting definition on Eclipse.

2.3 Integration with Webots simulator

Once we set the Eclipse working environment, we need to set its simulator as well. This simulator is provided by the Webots platform, already showed in the first chapter at section3, which execution run on a proper application. A straightforward way to make interoperable two different applications whose running on the same machine[2] is means means of TCP/IP sockets.

As depicted in fig.3.4 below, one of the artifact's main activities is to get raw sensor information in order to create symbolic information whose compose the agent's knowledge base. With this purpose, the artifact creates a new ServerSocket on a well-know port -at least, let's pretend that it is so- , waits for a connection request from the robot platform, and then when that happens, it opens a new socket with it. Through this TCP connection, executive artifact can receive raw data from sensors and issue robot commands whose are further transformed by the functional level in simple mechanical actions.

[2]This is our scenario, however there is no problem if we move the code of the functional layer (that interfacing the physical world) onto a different network node, since in the future such code will be execute on a real robot.

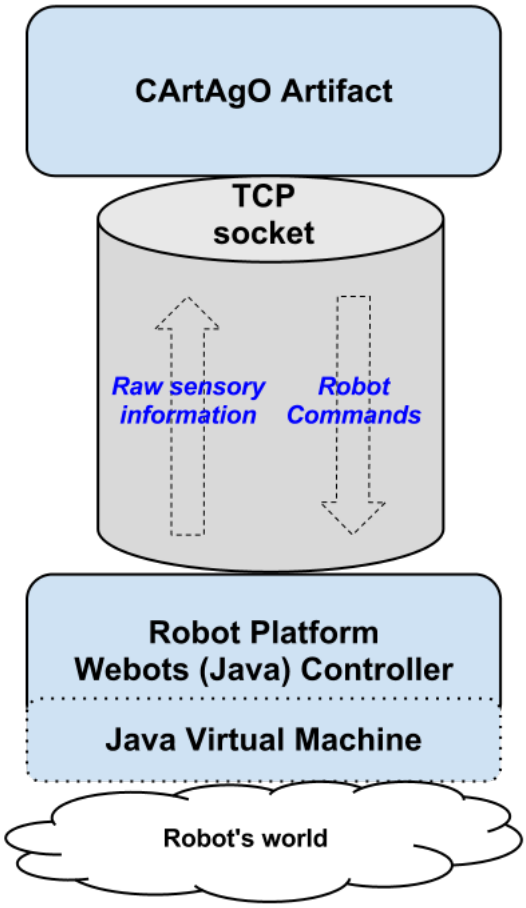

Figure 3.4: Interaction between the lower layers through TCP socket.

To facilitate the use of agent programming languages for developing robotic control systems, such languages should provide suitable interfaces to integrate with existing robot frameworks. These interfaces ideally provides built-in support for communication and control mechanism of robotic frameworks[23](e.g ROS seen in Chapter1). However in our explorations we are not going to use a specific robotic framework -even though it is possible using ROS / URBI along with Webots- but we will exploit the Webots APIs. Webots provides several APIs written in a handful of languages like C, C++, Matlab, URBIscript, Java and Python. These APIs allow programmers to interact directly with the robot with a set of methods -in Java for example- and to obtain sensor values.

Using these APIs instead of others commercial frameworks standard inter-

faces for accessing heterogeneous robotic hardwares, carry the same advantages in terms of facilitate robotic software development and reuse by using software engineering techniques such as component based software development.

It is worth it to point out what is exactly the robot platform that is interfaced with the artifact layer. Simply, it is the real **controller** of the robot. As a matter of fact, since Webots is meant to be used actually as a standalone robot simulator -though in this thesis we have moved the loci of control, to the agents platform, on Eclipse- we must to write the program that controls the robot, right here. Then, when a simulation starts, Webots launches the specified controllers, each as a separate process, and it associates the controller processes with the simulated robots[7]. This control code is compiled / interpreted or both of them (for that reason Webots run its own JVM).

2.4 Sensory Input & Actuator Commands

The controller just mentioned will be developed in Java[3], so it turns out necessary to define a class that will be tailored to represent information from sensors. It is composed of two parameters: the former suggests an integer representing the type of sensor data (e.g an image from camera, a double value from a distance sensor and so on), the former is the actual value of such sensor information. In AppendixA we show the SensorInfo Java class -with its methods to set up and modify sensor information to send-, which is the actual object that will be send (through readObject() Java method) and read through the socket.

In order to get every SensorInfo object we need in addition to define what in CArtAgO is named a *BlockingCommand*. Roughly speaking, in order to implement an artifact that provides I/O functionalities for interacting with the external world -like network communication in this case-, CArtAgO provides a kind of **await** primitive that accept an *IBlockingCommand* object representing a command to be executed. The primitive await(IBlockingCommand)

[3]This choice is due to our programming languages background and skills.

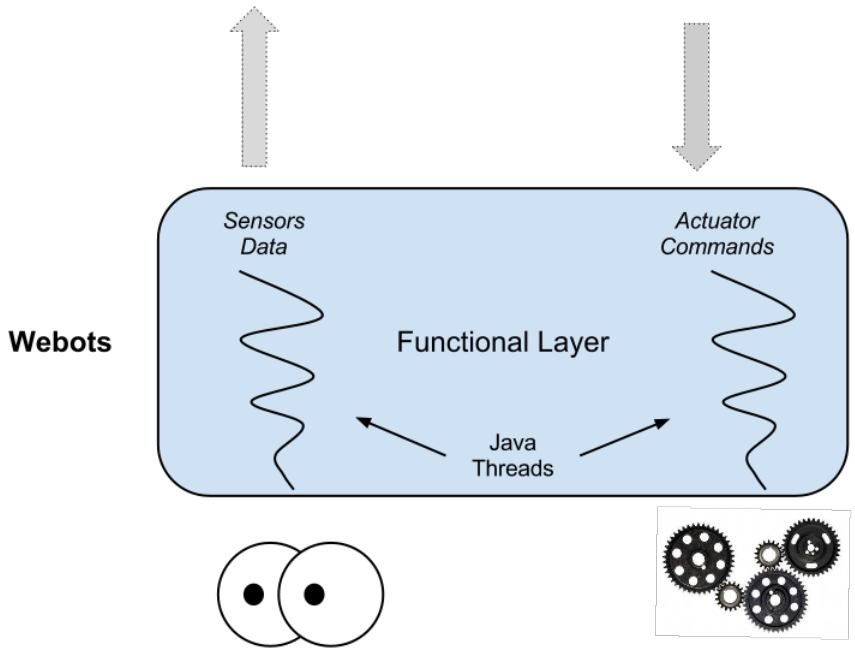

Figure 3.5: Parallel activities perfomed by functional level.

suspends the execution inside the artifact until the specified command - typically some kind of I/O operation with blocking behaviour, such as the readObject() method called by the artifact to receive a sensor data- has been executed. So, using that primitive, the artifact is awakened and can starts to process the received data.

It is worth remarking that, since the functional layer has both to continuously send sensory data and receive robot command, the robot platform controller is composed of two parallel Java thread whose run concurrently. Each thread executes one of the above activities, so as to do not suspend to send information when a new command has received through the same communication channel.

After talking about information that flows from the robot to the agent, we must specify how the system behaves according to the stream of commands addressed from the agent -and from the agent in turn- to the robot. We decided to employ the easiest approach to do this: the artifact simply send an integer number through the OutputStream. When the simulator side receive

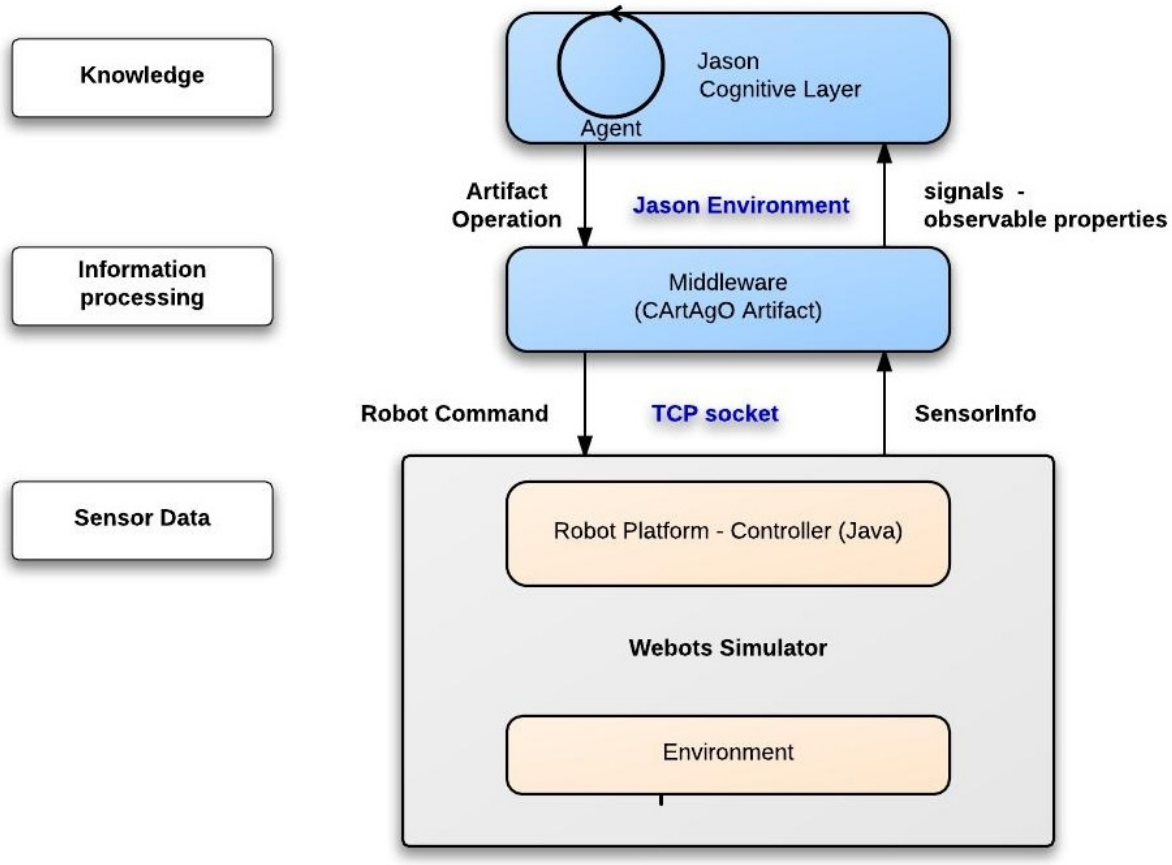

Figure 3.6: Final system layout. To achieve this result we drew on the work carried out by Wei and Hindriks in [25]

this integer, read its value and acts accordingly. Of course, initially both the sides have to agree on the means of each integer value (eg. the number zero could means "stop moving", the one "go forward" and so forth) coherently, otherwise several problems may arise.

In conclusion, we have encompassed every relevant aspects regarding the development of the software working environment that we set up, in order to be exploited for the experiments introduced in the next chapter. Now, as depicted in fig.3.6 we have a complete overview of the software system, along with its components and "joints" among them.

3 Recapitulation

Actually it is not so easy to extricate among the large number of applicable architectures for modelling the agent system. Thanks to the BDI model taken into account to define the nature of the agent which controls an autonomous system like a robot, we have been able to implement the software architecture of the system straightforwardly.

It is obvious that there would be a lot of work left to do if we want to enhance different aspects that concern what already done. Indeed, we have not gone so deep for what regards some technicalities like the low-level communication infrastructure or the customisation of Jason agent environment, its class or internal actions. Which are these future extensions and improvement will explain better in the -concluding- Chapter6.

Chapter 4

Experiments

In order to understand what reasoning about the organization and the design programming of a robot means, in this chapter we show four case studies involving a robot with a growing set of skills and then of complexity coefficient. We have exploited two kind of robots, both have two differential wheels (we can set different speeds to each one) situated inside an unknown dynamic environment we created before. The former has just two distance sensors situated on the front left and right whereas the latter, has five distance sensors (one in front of it and two for each side), a camera and a gripper. Finally, in the third example the robot is endowed with a GPS and a compass.

Firts, we provide a high-level description of the experiment we want to put into practice. Afterwards we give a general resolution strategy, as the set of behaviours the robot has to adopt, to achieve the above requirements, pointing out the critical aspects.

We make the effort to provide a resolving strategy *platform and language independent* as general as possible, as if we say it to another person and not to a robot:

the higher is the level of the description of such strategy the closer we get to the solution of the problem

making it also modular and extensible thanks to the BDI nature. Later, by exploiting the requirements and constraints description the robot has to abide by, we implement its controller using both with a structured paradigm

(like C programming language, commonly used in the robotics field) and an agent oriented language like Jason -plus CArtAgO.

As far as both implementation are concerned, we must say that a part is implemented aside from the specific example, so as to provide some guidelines about how sensor information can be mapped. On the other side, however, a part is implemented regarding the particular case study we are going to tackle.

Introduction

We believe it is worthwhile discussing about the interaction between the set of sensors (robot actual environment) and robot platform (the functional layer). Basically this happens by "polling" -through low-level APIs, provided by Webots- the sensors, in order to obtain their value continuously. Polling is employed when it is not possible to be notified by means of events about changes of a certain sensor -*event-driven* interaction. In this sort of interaction the concept of **interruption** is unknown, since we do not know when the sensor value changes.

Therefore the best pattern to obtain would have a basic layer -that is interfaced with the physical components- which allow us to handle even the above aspect. However, in Webots simulator the overall set of sensors is not designed to send notifications but just to provide its values, after an explicit request. So, we decided to adopt a classical polling approach, where robot's perceptions are mapped into agent beliefs. That is perceptions remain inside the knowledge of this agent, as the internal high-level representation of the external world.

This kind of approach, affects the design of the set of plans whose make up the agent "mind". In addition, some concerns have cropped up from the interaction with actuators and sensors, using a BDI-based agent control.

Let's suppose we have the action *"move forward for two meters"*. How do we know whether the robot has moved actually for that distance, since it provides only commands that enforce simple actions (e.g *"go ahead for one step"*?). To infer that the actions issued are performed effectively, we should check the perceptions about the world. This is a complex aspect of robotics field. An acknowledge from actuators would be useful for this purpose, but

unfortunately in our case, it is not possible to guarantee the effectiveness of all actions. Therefore two ways can be undertaken:

1. the agent loops continuously, requiring sensors value until a desired outcome is reached;
2. the agent delegate at the lower levels that cycling task, so that to be notified when a relevant condition of the world occurs.

We found the latter way more suitable according to thesis' aims. Thus we have opted to enrich the artifact -which we are going to call **Middleware Artifact** from now on- or rather, the executive layer, so as it carries out all the cyclic activities or those that entail a low-level data processing (e.g analyze the bitmap of a camera image). The artifact in turn, has to define what information model as *observable properties* -and then, the agent will be notified by means the changes of these- and what situation convey to the agent through *signals* -maybe warn the agent about critical scenarios.

Summing up, we are going to enforce the layered approach already discussed in the previous chapter. A layered architecture where some tasks will be delegated to the perceptive level (the middleware) letting the cognitive level to execute and exploit high-level activities and data. This is a fundamental aspect we must take into account for designing overall system.

Start the Simulation

Taking into account the software working environment presented in Chapter3, it is worth showing the set of steps to perform, in order to start the simulation using our software facilities.

Once we defined the agent in Jason and the Webots "world"[1] we are ready to start the whole system. First we have to open Eclipse application, go to our Jason project and then select the projectName.mas2j file, then click on *Run Jason Application*: this will start the agent environment (see fig.4.1).

Afterwards, we must open the **name.world** file inside Webots siumlator and just press the "play" button that get off the simulation. During the simulation we can stop it and get it up to thirty-times faster.

[1]This word is used in Webots, to express the environment in which the robot will run. In terms of lights, floor, robots, walls, physics parameters and so on.

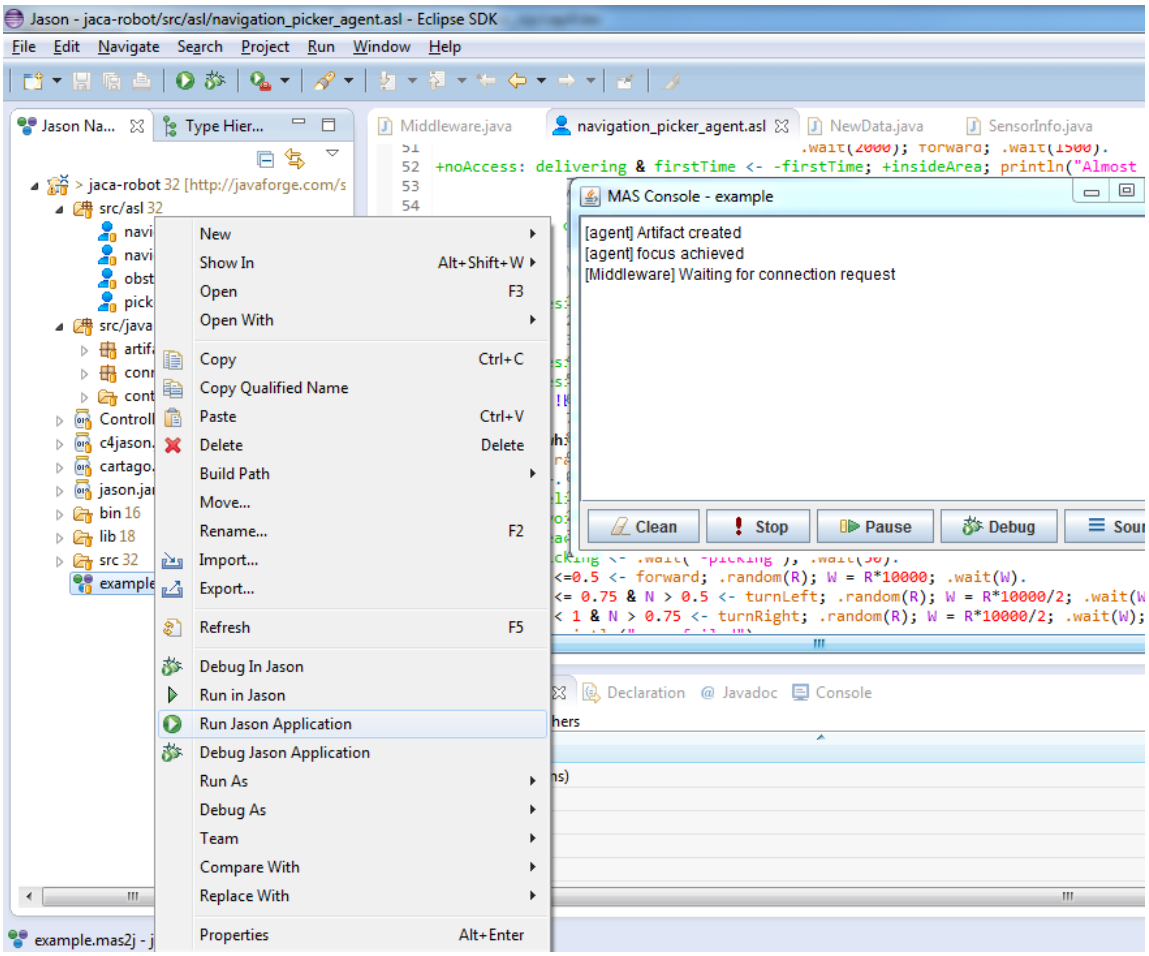

Figure 4.1: First step: starting the MAS.

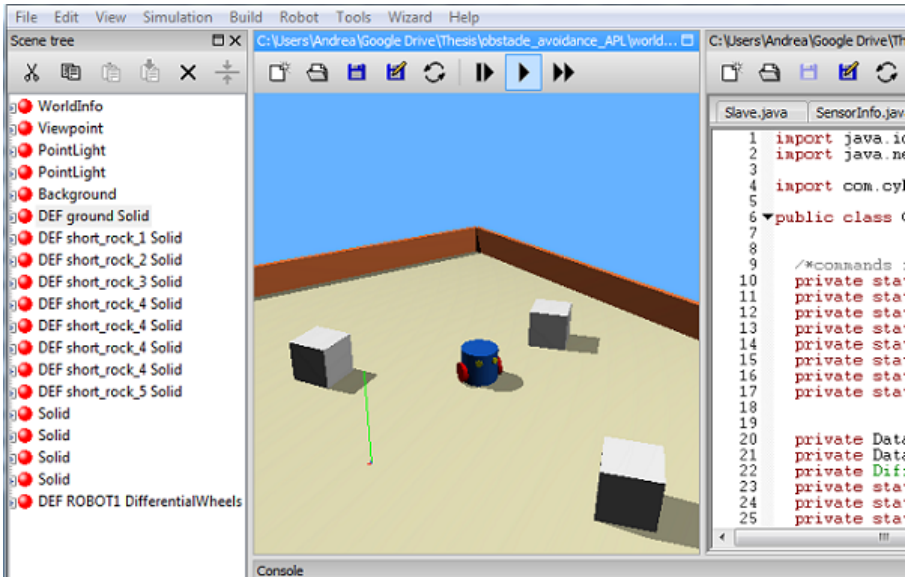

Figure 4.2: Second step: start the 3D simulation rendering.

1 Obstacle Avoidance

The robot used (see fig.4.3) has to move randomly within a bounded unknown environment, trying to avoid to bump against the cubic obstacles spread all over the floor and walls that form the boundary of such environment, exploiting the set of information from its distance sensors. Obstacles can be moved to different locations or even deleted from the environment, moreover is possible to add new ones.

1.1 Strategy

To move randomly, the robot must select from time to time, one of the three basic movements whose constitute the whole motion: **go forward**, **turn left**, **turn right**. Once it selects one of these, executes such movement for a casual period of time. While it is moving, if the difference between the distance sensor values exceeds a given limit, or in case at least one of those values is greater than the warning threshold, then an obstacle is detected. Therefore the robot has to stop random navigation and change the speed of both its wheels according to the maximum velocity allowed and the closeness of the obstacle. So that it will turn right/left -accordingly the obstacle is next

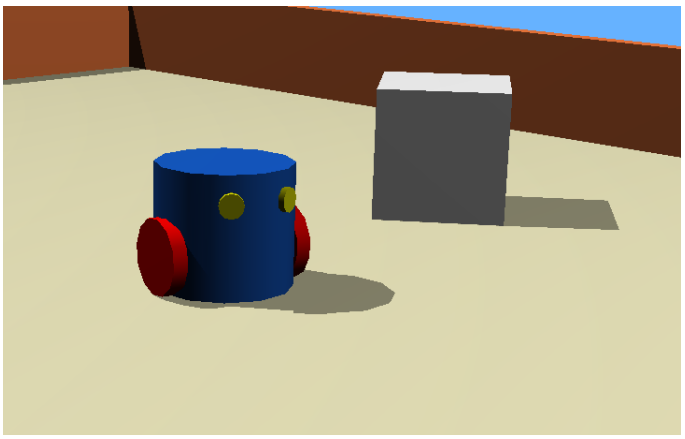

Figure 4.3: A simple differential wheels robot, endowed with two infrared distance sensors.

to its left/right side- until the obstacle is not detected anymore or, to be more precise, when distance sensors are back to safe levels. Then, it can restart the random motion.

1.2 Implementation

In this section -and in the following ones named in the same way- we will present briefly both the approaches used to implement the resolving strategy showed before.

Jason

In this case study, the robot is only interested about the detection of an obstacle, so as to undertake the "avoiding task". Since we want to delegate the activity which encompasses the constant request of distance sensor values, to the middleware, we have modelled the presence -and the absence- of the obstacle, as a observable property: obstacle(true/false). With regard to the change of such belief, that agent will trigger or stop the avoiding task.

The agent reacts to this belief updates, by triggering the plan hereunder when the perception of the world about the presence of an obstacle changes

```
+obstacle(true): delta(V) & not avoiding <-
                        +avoiding; println("obstacle: ",V); stop; .wait(50); !avoid(V).
```

This, starts the avoiding task, that send the "turn" command to the robot actuators (the differential wheels in such case), and mantains it until the obstacle is not detected anymore. In order to express that an obstacle is not present anymore, we have decided to exploit the wait operation, that suspend the plan execution until the specified event occurs: no obstacle, or rather obstacle(false) belief update event.

```
+!avoid(0) <- .print("Central obstacle"); backward; turnAround;
              .wait("+obstacle(false)"); -avoiding.
+!avoid(D): D < 0 & maxSpeed(M) <- setSpeeds(-M/2,M/2);
              .wait("+obstacle(false)"); -avoiding.
+!avoid(_): maxSpeed(M) <- setSpeeds(M/2,-M/2);
              .wait("+obstacle(false)"); -avoiding.
```

It is worth taking a deeper look into avoiding belief. We have already remarked that an important goal of this thesis is to get a modular solution, thus we think that using a belief-based suspending behaviour between subtask, would be an effective way to extend the agent's mind with slight changes when it increase its skills. In this case study, such belief represent whether the avoiding task is running or not, and -as we see below- if so, the moving task will be suspended until the avoiding activity is terminated successfully. The termination of an activity is meant to be modelled through its *belief removal*. In this way, we can also define a sort of priority between the set of subtasks. Indeed, we can convey that the avoiding task is more important than the moving activity by means of this plan:

```
+!move(N): avoiding <- .wait("-avoiding"); .print("avoided"); !move.
```

Therefore, the robot suspend the random navigation if the avoiding task has been triggered, and restarts moving when this belief is remove -the task is terminated.

After seeing how to map the strategy mentioned previously, into a Jason agent, now we can show the implementation of such strategy according to the artifatc aspects. To define the observable property that will be updated when a obstacle is detected, we wrote in the **MiddlewareArtifact**:

defineObsProperty("obstacle",false);

then, we have defined an internal action that loops until a boolean guard (stopAcquire) is switched to false. This internal action analyzes each information from the robot platform, and according to their values, updates the agent beliefs; below we show a snippet of middleware code:

```
@INTERNAL_OPERATION
void servingLoop(){
    double[] values = new double[]{};
    int type;
    double delta,right,left,hleft,hright, central,force = 0, limit = 0.15;
    stopDistance = false;
    while(!stopAcquire){
        await(data);
        info = data.getInfo();
        type = info.getId();
        switch (type) {
        case RIGHT_DISTANCE_SENSOR:
            if(!stopDistance){
                right = info.getVal();
                left = getObsProperty("leftSensor").doubleValue();
                getObsProperty("rightSensor").updateValue(right);
                delta = left - right;
                getObsProperty("delta").updateValue(delta);

                if(delta > 100 | delta < -100){
                    getObsProperty("obstacle").updateValue(true);
                }else if(getObsProperty("obstacle").booleanValue())
                    getObsProperty("obstacle").updateValue(false);
                if(right > 350)  getObsProperty("obstacle").updateValue(true);
                if(right < 200 & left < 200 & getObsProperty("obstacle").booleanValue())
                    getObsProperty("obstacle").updateValue(false);
            }
        }
```

The internal action is started by the agent, through the CArtAgO operation acquireData defined inside the artifact with:

```
@OPERATION
void acquireData(){
    print("Acquiring data");
    execInternalOp("servingLoop");
}
```

As showed in the avoiding plans: turnAround, backward and setSpeed, are three high-level robot commands -pretty explicit- whose are provided by the **MiddlewareArtifact** in order to be issued to the real robot through the

TCP socket. These operations are defined as (we give just one of them as an example):

```
@OPERATION
void setSpeeds(double x,double y) throws IOException{
    out.writeInt(SET_SPEED);
    out.writeDouble(x);
    out.writeDouble(y);
}
```

C

Using the C programming language, we do not have the notion of event, so we must use a set of *if* statement in every critical point of the program, so as to have an updated perception of the world before any other operation. With this approach, we are going to model the avoid activity by means of a proper function that is triggered in case sensor values exceed certain limits, as we mentioned early. The avoiding function is so defined:

```
static void avoid(double value){
  if(value < 0){
     wb_differential_wheels_set_speed(-MAX_SPEED/2, MAX_SPEED/2);
  }else{
     wb_differential_wheels_set_speed(MAX_SPEED/2, -MAX_SPEED/2);
  }
  while(obstacleDetected){
     leftSensor = wb_distance_sensor_get_value(dsL);
     rightSensor = wb_distance_sensor_get_value(dsR);
     centralSensor = wb_distance_sensor_get_value(dsC);
     if((leftSensor < 350) & (rightSensor < 350) & (centralSensor < 350)){
       obstacleDetected = false;
     }
   wb_robot_step(TIME_STEP);
  }
 wb_differential_wheels_set_speed(50, 50);
}
```

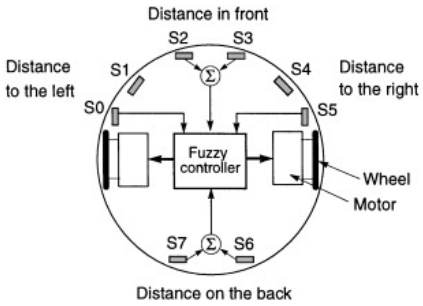

Figure 4.4: Technical representation of khepera robot, and its sensors.

2 Object Picking

In this example, the environment is the same seen above, with a red cylinder more that can be also moved within the floor. Besides the random move, the obstacles/walls avoidance, it exploits the distance information from the central distance sensor: if the robot detect an obstacle thanks to the central sensor starts to analyze the images from camera (in particular, the central section of them), in case it detects a red cylinder[2], then it takes and lifts the cylinder with its gripper and keeps on the random motion through the obstacles, otherwise avoids the obstacle detected.

Additional Requirements

While the robot is trying to pick up the cylinder undersired situations may occur. For instance, the cylinder is moved after its detection or the mechanical arm could not be able to raise after he grabbed the cylinder. Moreover, once the robot has retrieved the cylinder, it must be warned if the cylinder fall from the grips: in such case, it terminates both the running and pending actions and trasmits the problem through a message. These situations could happen because of the dynamic nature of the environment and all its elements: so the robot has to face such failures, in order to ensure a good level of safeness.

Additionally, in case the robot identifies an obstacle with lateral sensors, then it checks wheter the red is the predominant colour. If so, we assume

[2]Actually we are not able to tell whether the solid is exactly a cylinder, but in this example we thought that this point was not considerable for thesis' purposes.

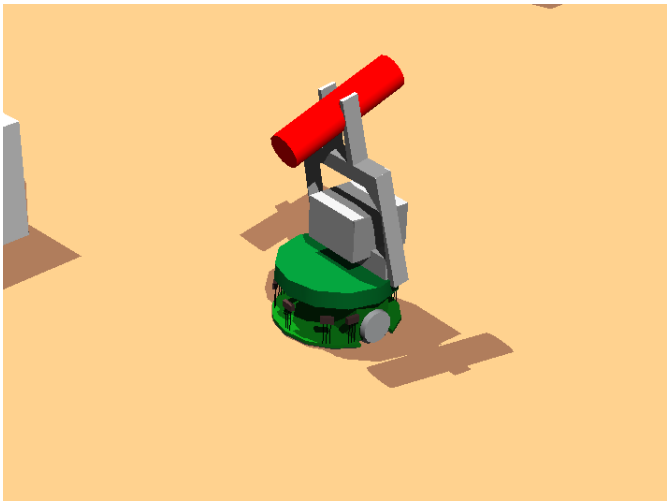

Figure 4.5: A Webots view of the khpera robot -equipped with a gripper- used in the next experiments.

that object is a red cylinder, so the robot stops and tries to turn so that it can pick the cylinder up.

The mechanical systems used in this experiment is the Khepera robot (see fig.4.5) a commercial robot, broadly distributed in university research centers.

2.1 Strategy

When the robot detect an obstacle (the strategy adopted to avoid the encountered obstacles is the same mentioned in 1.1) with the central distance sensor, it analyzes camera images comparing the pixel red levels with the green and blue ones. If there are not a ruling colour component, then the robot has met an obstacle -whatever- and simply avoids it. Whereas, if the red component results at least three times as much as the others, then it has detected a red cylinder. So, it suspends the random navigation, opens and brings down the gripper, finally it closes the gripper at a given values, expressed in centimeters.

If the grip does not reach a certain value -that indicates in terms of pressure, whether the cylinder is grasped- by a safety lapse of time, then the

cylinder is assumed moved to another position or fallen: thus the robot opens the gripper again, turns around itself and resumes the random navigation.

If the cylinder is correctly grabbed, the robot rises its mechanical arm and waits its arm reaches a desired position within a certain period of time: when the cylinder is actually picked up, the robot can resume the previous navigation inside the enrvironment.

Even in the case the robot detect an object next to its left/right, before performing the obstacle avoidance task, starts to analyze the red levels of the lateral sections of camera images. If red is the predominant colour, that means it is close to a cylinder, so it stops and turns according to cylinder position, until such cylinder is detected by the central distance sensor so that it is possibe to enforce the procedure for retrieving, just described. Once the red cylinder is picked up, if the grip pressure become too low, then the robot assumes the cylinder is fallen from the gripper and stops the navigation and comunicates the problem.

2.2 Implementation

In this experiment, the robot must carry out two more significant task. Each of them is concerns the activity of picking up the red cylinder, with regard to the position of such cylinder, either lateral or in front of the robot.

Jason

We need now to define suitable plans to fulfill the activities described in 2.

To do that, decided to model the notification of the closeness of a red cylinder by means of a signal, cylinderDetected, to which the agent reacts so as to perform the picking operations.

```
+cylinderDetected: not picking <- +picking; stop; stopAcquireDistance;
                                  .wait(50); catching; !block.
```

Furthermore since the robot has got the ability to reach a lateral cylinder, we have modelled such aspect similarly as just seen, through signal from the perceptive layer (the artifact).

```
+cylinderLeft: reachCylinder(false) & not insideArea & not charging
                        <- -+reachCylinder(true); stop; setSpeeds(0,10); !reach.
+cylinderRight: reachCylinder(false) & not insideArea & not charging
                        <- -+reachCylinder(true); stop; setSpeeds(10,0); !reach.
+cylinderBeyond: reachCylinder(true) & maxSpeed(M)
                        <- turnAround; .wait(20000); setSpeeds(M/5,M/5); !reach.

+!reach: reachCylinder(true) <- .wait("+cylinderBeyond",5000).
+!reach: reachCylinder(false) <- .wait("+cylinderDetected",3000).
-!reach <- .print("Reach lateral cylinder failed."); -+reachCylinder(false).
```

The catching artifact operation, is meant to represent the picking operation by means of the mechanical arm, while block achieve goal, starts all those plans that check the reliability of each single operation, according to time constraints.

As before, we need those that we call ***coordination beliefs***, in order to coordinate the running subtasks and get a well-engineered system. Thanks to picking and reachCylinder(true/false) the agent knows whether one of these activities has been triggered. To give them a higher priority than the navigation one, we apply a simple plan that suspend the random wandering:

```
+!move(N): picking | reachCylinder(true) <- .wait("-picking"); .wait(50).
```

The **MiddlewareArtifact** checks constantly the value of the gripper pressure on the cylinder, and in case this is too low, the agent is warned by a proper signal. The agent in turn enforce the following plan

```
@fallCylinder[atomic] +fallen: picked(true) <-
                                .drop_all_intentions; .print("Cylinder has fallen.");
                                 stopAcquire; stop.
```

which is annoted as an atomic plan, so that any other activities will not be triggered until that plan is not completely carried out. Therefore this should bring a right level of safety, since we believe that this controller is well-suited for a real robot too.

Finally, because of while the robot is picking up the cylinder, the mechanical arm is detected as an obstacle and undesired behaviours could come out, we have to apply a tiny modification to the plan that reacts to the obstacle detection, so as to avoid these behaviours.

```
+obstacle(true): delta(D) & not avoiding & not picking & reachCylinder(true)
                                <- +avoiding; stop; .wait(50); !avoid(D).
```

In this way whatever is near to the robot, is not considered as an obstacle, as long as the robot is picking up a cylinder or is turning slowly fot gettin closer to it.

Here, the observable properties that the artifact defines, and the agent will use are:
defineObsProperty("picked",false)
and
defineObsProperty("caught",false)

Exploiting the **JaCa** nature, we delegate the heavy operation concerning the image processing to the lower layer, which notify the agent, if the obstacle met is a red cylinder or a ordinary obstacle to avoid. Such activity, has been implemented like the snippet below shows:

```
case CENTRAL_DISTANCE_SENSOR:
    if(!stopDistance){
        left = getObsProperty("leftSensor").doubleValue();
        right = getObsProperty("rightSensor").doubleValue();
        central = info.getVal();
        getObsProperty("centralSensor").updateValue(central);
        double red,blue,green;
        if(central > 480){
            red = rgb[0];
            green = rgb[1];
            blue = rgb[2];
            if(red > 2*blue & red > 2* green){
                lateralCylinder = false;
                signal("cylinderDetected");
            }
            else getObsProperty("obstacle").updateValue(true);
        }
        if(right < 200 & left < 200 & central < 200
                & getObsProperty("obstacle").booleanValue())
            getObsProperty("obstacle").updateValue(false);
    }
    break;
```

C

For developing the same behaviour with C we have implemented in the controller class, new operations and functionalities, like image processing and cylinder picking. These task are defined as C funtions and we show hereunder the pick up function, focussing on how it deals with time constraints

```
    ****
  while( !(force > -500 && force < -5) & (timeout > 0)){
    wb_robot_step(TIME_STEP);
    force = wb_servo_get_motor_force_feedback(left_grip);
    timeout = timeout - 50;
  }
  if(timeout == 0){
    printf("Object not found");
    wb_servo_set_position(servo, -1.4);
    wb_robot_step(TIME_STEP*20);
    /* turn around and go ahead */
    wb_differential_wheels_set_speed(-50, 50);
wb_robot_step(TIME_STEP*40);
    wb_differential_wheels_set_speed(50, 50);
    return 1;
  }else{
    /*..keeping on with picking operations...*/
    ****
```

Even though there is no so many features to add to the robot, we must add to the main loop of the controller, a handful of *if* statement in order to not miss any relevant sensors situation. Considering that the most critical scenario that can occur is the one when the picked cylinder falls from the gripper, the program flow needs to check first of all if the pressure value is enough

```
    if(picked){
      if( !(force > -500 && force < -5) ){
        picked = false;
        /* stop the robot */
        wb_differential_wheels_set_speed(0, 0);
        printf("Cylinder has fallen. Anybody retrive it.");
```

```
      break;
    }
  }
```

then, when it detects an close object, starts the image processing so as to determine whether perform either the picking task or the avoiding one

```
***
if(centralSensor > 480){
    image = wb_camera_get_image(camera);
    /*isRed represents the image processing
    function*/
    if(isRed(image)){
    /*if red is the predominant color, pick up..*/
      printf("Cylinder found");
      wb_differential_wheels_set_speed(0, 0);
      int res = pickUp();
      if(res < 0){
        printf("error encountered");
        break;
      }else printf("Picking ok");
    }else{
      /*..otherwise is an obstacle, avoid it!*/
      obstacleDetected = true;
      avoid(delta);
    }
  }
***
```

3 Navigation

This robot perform the same set of tasks mentioned in the previous examples, furthermore after the cylinder is retrieved, the robot brings it to a well known point to which the robot wish to move, by means of the GPS sensor and compass onboard without an environment's map, just trying to follow the straight path between the current position and that point (we suppose this is the shortest path without considering the obstacles), keeping on avoiding

obstacles. After that it starts to move randomly again, in order to find and deliver another red cylinder.

Additional Requirements

Once the cylinder is delivered to the target point, the robot must leave the area next to that point before undertake the random navigation again, and it cannot come back there unless it is bringing another cylinder.

3.1 Strategy

In this exploration the strategies to avoid obstacles and pick up cylinders, are the same seen above. Moreover when the cylinder is actually retrieved, in order to move towards the well-known gathering place, the robot must align itself. To do that it gets its orientation -representing the current direction compared to the North- thanks to the digital compass onboard and computes the desired direction it must obtain so as to try to reach the destination moving along a straight direction[3].

This is possible by exploiting the well-know location of the target point and the current robot position, by means of a GPS sensor -whose the robot is equipped with. The difference between current and desired orientation is the gap to nullify. If this gap is included between 0 and -180 degrees or it is greater than 180 degrees, then the robot has to turn right, otherwise to turn left, until the compass reach the desired value (or, at least in the range of that) and the gap is close to zero, so that the robot can start moving straight. If the robot encounter an obstacle, enforce the usual avoiding task then, once this is avoided, calculates the gap again and repeats previous operations to align itself correctly.

In figure4.6 we show a rough representation of the likely heading of the robot, comparing x-axis which that points the North. From that we obtain the current robot heading α and the desired heading β, whose difference define the gap to fill in degrees δ. We can calculate it as:

$$\delta = \tan^{-1}(\Delta Y/\Delta X) = \tan^{-1}((Yg - Yr)/(Xg - Xr))$$

[3]Since we do not know nothing about the environment, the simpliest way to get to a certain point, is to move in a straight line towards it.

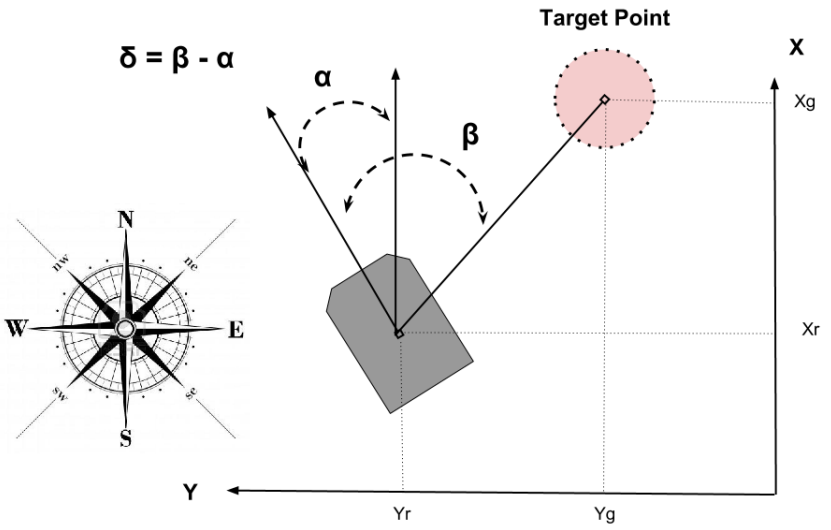

Figure 4.6: Xg/Xr and Yg/Yr indicate the global position of the location the robot wishes to head for and the position of the robot respectively.

When it is nearby the gathering point, stops the navigation, brings down its mechanical arm, opens the gripper to release the cylinder, rises the arm and finally turns around and moves foward so as to exit from the gathering area. It will be able to restart the random navigation when it will be out of the gathering area, comparing its position (obtained from GPS) with the circular area.

3.2 Implementation

It is worth remarking that the robot navigation within a dynamic environment is anything but an easy problem to solve. This task is even much more complex considering that the robot does not have any map of the environment and knowledge about the position and the shape of them. Therefore our implementation has not explored all the possible investigation concerning the related problematics like accuracy of gathered data from sensors, and their precision. Even though we are not going to face every issues of robot navigation, we have defined a clear method tailored for our purposes, in order to head the robot for the target point.

Jason

The gathering point is the only thing that the robot knows a priori, thus a clear way to model this awareness is to map that point as a belief already present in the agent belief base.
targetPoint(x,y).
where x and x represent the position in the global (world) coordinate system.

When the picking operation is successfully completed, the robot aligns itself in order to heads towards the target point. To do this, first computes the gap (computeGap) from the right alignment, getting the desired orientation in degrees and then starts the plans (balance) whose according to the current and the desired orientation, issues right turning commands to the robot until desired orientation is reached -around a certain range.

```
+!go_ahead: picked(true) <- println("Cylinder picked up"); -picking; +~delivered;
                            +delivering; !computeGap; !balance; forward.
```

```
+!computeGap: orientation(Current) & location(X,Y,Z) & targetPoint(Tx,Tz)
                                    <- computeAngle(X,Z,Tx,Tz,Desired);
                                    -+desired(Desired).

+!balance: desired(D) & orientation(O) & O > D-0.6 & O < D+0.6
                                    <- stop; -~around; .drop_intention(balance);
                                    .drop_intention(computeGap).
+!balance: desired(D) & orientation(O) & ((O-D >= -180 & O-D < 0) | (O-D >= 180))
                                    <- setSpeeds(25,-10); .wait(20); !balance.
+!balance: desired(D) & orientation(O) & (O-D < 180 & O-D > 0 | O-D < -180)
                                    <- setSpeeds(-10,25); .wait(20); !balance.
+!balance <- !balance.
```

Where delivering and ~delivered are the ***coordination belief*** whose say that the delivering task is currently running and the cylinder is not delivered yet. Aftwards, when the robot is heading towards the right direction it has not to ignore other cylinders, nevertheless the avoiding obstacles task is still alive. This is a feasible operation to enforce by exploiting the **coordination belief** so as to indicate the priority among several plans.

```
+cylinderDetected: not picking & not delivering
                                <- +picking; -reachCylinder(_); +reachCylinder(false); stop;
                                   .print("Cylinder detected"); catching; !block.
+obstacle(true): delivering & delta(D)
                                <- .drop_intention(balance); +avoiding; stop;
                                   !avoid(D); forward; !computeGap; !balance; forward.
+!move(N): delivering <- println("Waiting delivering"); .wait("-~delivered");
                         .print("Delivered!"); !move(N).
```

In such way, when the robot encounters an object -either a cylinder or not- the avoiding plans are triggered, to avoid that object, thereafter that plan restarts those plans whose fill the orientation gap, in order to align the robot to the destination. When the destination is reached, then the agent starts the task which involves to put down the cylinder it is carrying on and to turn around and go forward so as to get out the target area.

```
+storageReached: delivering
            <-  .drop_desire(balance); .drop_desire(computeGap); .drop_desire(avoid);
                stopAcquireDistance; -delivering; .wait(100); .print("TARGET REACHED");
                stop; .wait(40); putDown; -desired(_); ?orientation(O); D2=O+180;
                +desired(D2); !balance; .print("balanced");
                forward; waitExit; .wait("+~around"); -~delivered.
```

There are two observable properties that the artifact defines, that turn out fundamental for the execution of the agent: the former indicates the global position of the robot obtained thanks to the GPS onboard, the latter provides the current orientation -expresses in degrees- of the robot according to the North.

```
defineObsProperty("location", 0.0,0.0,0.0);
defineObsProperty("orientation", 0.0);
```

Finally, when the artifact detects that the robot has reached the target point, notifies the agent through a proper signal to which it reacts applying all the necessary operations to put down the cylinder and terminates the delivering activity. We have reported below the snippet of artifact -within the internal action loop- which checks whether the robot has reached the destination and the resulting agent reactive plan.

```
case LOCATION:
    values = info.getCoord();
    if(!(values[0] == 0 & values[1] == 0 & values[2] == 0)){
        coordinates[0] = trunc(values[0], 2);
        coordinates[1] = trunc(values[1], 2);
        coordinates[2] = trunc(values[2], 2);
    }else coordinates = new double[]{0,0,0};
    if(targetPoint != null){
        if( (coordinates[0] > targetPoint[0] - threshold) &
                (coordinates[2] < targetPoint[2] + threshold) &
                (coordinates[0] < targetPoint[0] +threshold) &
                (coordinates[2] > targetPoint[2] - threshold) &
                !targetReached){
            targetReached = true;
            print("Target reached");
            signal("targetReached");
        }
    }
```

C

The operations introduced in this experiment, like put down the cylinder, calculate the gap between current and desired orientation and balancing the fill this gap, are modelled as functions. We are going now to show the balancing function, in order to make later a comparison with the agent-based approach.

```
static void balance(double D,double C){
  if( ((C-D >= -180) & (C-D < 0)) | (C-D >=180) ){
     wb_differential_wheels_set_speed(15, -5);
  }else if( ((C-D < 180) & (C-D > 0)) | (C-D < -180) ){
    wb_differential_wheels_set_speed(-5, 15);
  }
}
static void balanceFunction(double D, double C){
  balance(D,C);
  const double *orientation = wb_compass_get_values(compass);
  compassX = orientation[0];
  compassZ = orientation[2];
  double currentAngle = convertBearing(orientation[0],orientation[2]);
  while(!(currentAngle > D - 0.6) & (currentAngle < D + 0.6)){
    wb_robot_step(TIME_STEP);
    currentAngle = convertBearing(orientation[0],orientation[2]);
  }
}
```

Where convertBearing is the function to get the orientation in degrees.

In the main control loop, once the cylinder is actually picked up, the balancing and the consequent delivering activities start. This will be expressed through the switching of two boolean values: fixingAngle and delivering.

```
if(fixingAngle){
  if( (currentAngle > desiredAngle - 0.6) & (currentAngle < desiredAngle + 0.6) ){
    printf("Desired angle reached, go!\n");
    fixingAngle = false;
    delivering = true;
    wb_differential_wheels_set_speed(30, 30);
    wb_robot_step(TIME_STEP);
  }else{
    //still turning
    balance(desiredAngle,currentAngle);
  }
}
```

In case, the delivering task is currently running, the first condition to check, is whether the robot is close to the gathering point. This may be implemented as follows

```
   ***
if(delivering){
   if((position[0] > storagePointX - THRESHOLD) & (position[0] < storagePointX + THRESHOLD)
   & (position[2] > storagePointZ - THRESHOLD) & (position[2] < storagePointZ + THRESHOLD)){
       printf("Storage point reached - Current pos %f %f\n",position[0],position[2]);
       wb_differential_wheels_set_speed(0, 0);
       /* turn around and go ahead */
       putDown();
       fixingAngle = false;
       wb_differential_wheels_set_speed(-MAX_SPEED/2, -MAX_SPEED/2);
       wb_robot_step(TIME_STEP*4);
       balanceFunction(currentAngle+180,currentAngle);
       wb_differential_wheels_set_speed(MAX_SPEED/2, MAX_SPEED/2);
       delivering = false;
       goingOut = true;
       printf("Cylinder delivered! \n");
     }
   else{
   ****
 }
```

```
  /*check whether there is an obstacle nearby*/
```

Since we do not want that the robot pick up cylinder while it is delivering the picked one, it is fundamental to specify inside some existing *if* statement whether the delivering task is already started. For instance:

```
    ***
if(centralSensor > 480){
        image = wb_camera_get_image(camera);
        if(isRed(image,CENTRAL_PART) & !delivering){
        ***
```

4 Task suspend/resume

This example is similar to the last seen showed, the only change concerns when the robot's battery level decrease beneath a safe threshold, the running activity is suspended so that the robot can steer towards the charger. Like the delivering task, the robot has to avoid the obstacles that could meet along such path. When it gets to the charger, it stays there for a while in order to recharge its battery and then comes back to that point where the running activity has been suspended, afterwards it resumes it.

4.1 Implementation

Jason

The charging point is already present inside the agent belief base likewise the gathering point
charger(x,y).
As we seen so far regarding Jason implementations, the charging task is identified as a belief, so as to get the highest priority.

This belief is added to the context of every relevant plan so that their execution become suspended and wait for the chargin belief to be deleted, for instance:

```
+!move(N): charging <- .print("wait recharging"); .wait("-charging"); !move(N).

+!pick: charging <- .print("Picking suspend, wait recharging.");
                    .wait({-charging}); .print("Picking resumed"); !pick.

+!go_ahead: charging <- .print("Delivering suspend, wait recharging.");
                        .wait({-charging}); .print("Delivering resumed"); !go_ahead.


+cylinderDetected: not picking & not delivering & not charging
                            <- +picking; -+reachCylinder(false); stop;
                              .print("Cylinder detected"); catching; !block.
```

the last one suggests that in case the robot is moving to recharge its battery, ignores all the cylinders whose could be encountered.

The middleware check continuously the battery level and when this become too low,

```
case BATTERY:
    battery = info.getVal();
    getObsProperty("battery").updateValue(battery);
    if(battery < 480 & battery > 0){
        signal("lowBattery");
    }
```

sends agent a proper signal which means that the robot is likely to run out the energy soon. The agent reacts accordingly to what explained in the resolving strategy:

```
+lowBattery: location(X,Y,Z) & charger(Cx,Cz) & not charging & orientation(O)
        <- mark; +charging; ?battery(B);  stop; -+targetPoint(Cx,Cz);
            setUpTargetPoint(Cx,0,Cz); !computeGap; !!balance; .wait({+targetReached});
            stop;.wait(3000); .print("RECHARGED"); ?battery(B2);
            -+targetPoint(X,Z); setUpTargetPoint(X,Y,Z); !computeGap; !!balance;
            .wait({+targetReached});
            stop; ?storagePoint(Tx,Tz); -+targetPoint(Tx,Tz);
            setUpTargetPoint(Tx,0,Tz); !restoreOrientation(O); -charging.
```

Where the agent reuses the same procedure to reach a certain destination, used to head for the gathering point. This is possible as long as we modify temporarily the arguments of the targetPoint belief. That's why the **-+targetPoint(Cx,Cz)** action: previous target point is updated with charger point coordinates in the belief base and afterwards inside the artifact, with setUpTargetPoint operation. So, the agent waits that the charger

is reached so as to try to move towards the point where the signal cropped up exactly as before. In addition, with the achieve goal !restoreOrientation(O) the robot try to assume the orientation it had before the task suspension.

```
@restore[atomic]+!restoreOrientation(D)
                    <- -+desired(D); !balance; stop; .drop_intention(balance);
                        .print("Restored"); .wait(500).
```

Once the charging task is terminated, we need to check whether the delivering task was running since it is th very neaxt activity to perform -according to the priority. For this purpose, we can implement a plan where the triggering event is the charging belief deletion

```
-charging: delivering <- .print("Restart delivering"); !computeGap; !balance.
```

In AppendixB it is possible to see the complete Jason implementation of the robot controller, which includes all the task modules discussed so far.

Behaviour-based approach

Due to the lack of time, we are not going to implement actually the behaviour-based strategy for this last experiment, however we are going to present a hypothetical architecture of the system, pointing out the most significant aspects that such approach encompasses. The first operation is the one concerning the individuation of the different behaviour modules whose compose the overall robot behaviour.

Behaviour modules are:

- **Obstacle Avoid**
- **Random Motion**
- **Cylinder Picking** which entail reaching lateral cylinders.
- **Reach Point**
- **Charge**

Each module entails several actions to be execute so as to fulfill its activities, following somehow a strategy. This strategy could encompass a fragment of the whole resolving strategy defined at the beginning of every experiment section. With regard to the **Reach Point** module, the behaviour programmer, seeing the resolving strategy already explained, could get an high-level strategy like:

```
Behaviour ReachLocation
  /*obtain the robot location*/
  robot_loc = getGPS_xy()
  /*calculate the distance between the points (dest_loc) is the well-known destination*/
  gap = dest_loc - robot_loc
  /*calculate the desired heading*/
  theta = arctan(gap)
  /*get the current heading*/
  heading = get_compass_heading()
  if(gap /= 0){
      /*destination not reached*/
     rotation = computeDirection(heading - theta)
     enforce(rotation)
  }
end behaviour
```

Below, a behaviour diagram (fig.4.7) can help us to figure out the type of sensors data, the modules need to use in order to perform their activities, and which behaviours can potentially conflict. Considering that, the role of arbiter is fundamental in this kind of systems -and in the robotics field- because it/they has/have to resolve the issue concerning which command the robot should obey. Of course, in non-trivial applications, there is the need to employ more than one arbiter.

The arbiter has to be well-engineered, so as to get the desired behaviour and be coherent with the early strategy definition as well as the additional requirements/constraints. In particular, it has to coordinate correctly the different behaviours, so that the most critical one, for example the charging behaviour, will suspend which are the currently running, in order to be executed immediately. After that, the arbiter will resume the behaviour suspended, trying to avoid any types of conflicts.

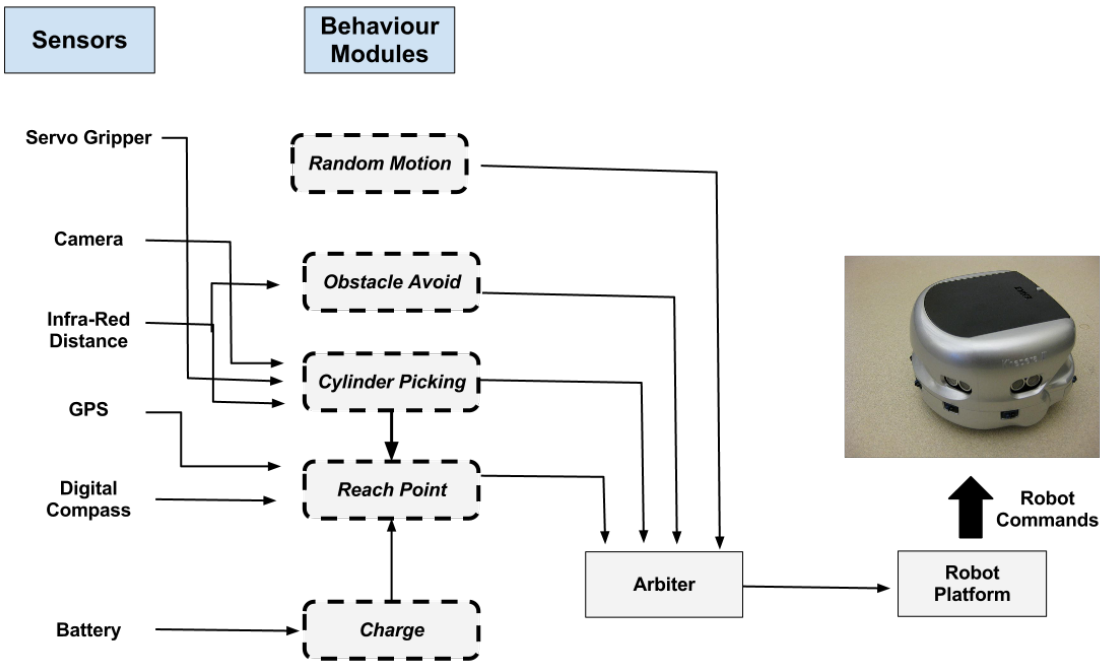

Figure 4.7: This behaviour diagram represents a possible behaviour-based approach to face the last experiment.

In this paradigm, rather than having a planning capability or an explicit goal-oriented behaviour, a robot's behaviour is emerged as the result of the behaviour of its own reactive componentes[23]. Although behaviour-based robotics has shown to be successful in many applications, it has been argued that such an approach is incapable of scaling up to human-like intelligent behaviour and performance.

5 Common Aspects

Throughout the experiments, few aspects are shared among the different agents implemented regardless the specific case study. One of these, is the procedure to create the artifact and put the focus on it, enabling this to accept socket connection requests, we can call it, the initial phase.

```
+!init(Id) <- lookupArtifact("middleware",Id); focus(Id).
-!init(Id) <-  makeArtifact("middleware","artifacts.Middleware",[4444],Id);
               println("Artifact created").

+!waitConnection: connected(true)
                        <- println("Connection successfull, waiting for sensor data");
                            acquireData; !moving.
+!waitConnection <- .wait(100); !waitConnection.
```

It is worth poiting out that connected(true) is an artifact-defined observable property which suggests the socket connection status, and accept is the artifact operation which open the server socket on the local host, waiting for connection requests. There are not C implementation of this phase because the whole robot controller runs on the Webots platform, so there is no need to connect to another framework.

Another common functionality concerns the random navigation, and below we show Jason and C implementation fragments of code: the second approach enforces this task as a function.

Jason

```
+!move(N): N <=0.5 <- forward; .random(R); W = R*10000; .wait(W).
+!move(N): N <= 0.75 & N > 0.5 <- turnLeft; .random(R); W = R*10000/2;
                                  .wait(W); forward; .wait(1000).
+!move(N): N < 1 & N > 0.75 <- turnRight; .random(R); W = R*10000/2;
                                  .wait(W); forward; .wait(1000).
-!move(_) <- println("move failed").
```

C

```
static void move(){
    /* choose a random move */
    double m = frand();
    if( m <= 0.5 ){
      /* forward */
      wb_differential_wheels_set_speed(50, 50);
    }else if ( (m > 0.5) & (m <= 0.7)){
      /* turn left */
      printf("turn left \n");
      wb_differential_wheels_set_speed(20, 50);
    }else{
      /* turn right */
      printf("turn right \n");
      wb_differential_wheels_set_speed(50, 20);
    }
}
```

Chapter 5

Considerations

Starting from the work accomplished so far, in this chapter we are going to gather and discuss benefits, drawbacks and considerations in general, by making a comparison between the utilization of a BDI approach to robot programming and a classic C-based approach. In particular an evaluation between the outcomes will be reported, in order to show clearly which aspects concerning robot programming, are positive and which, turn out to be not so good.

1 Evaluation of both the approaches

In this section we are going to make a comparison between the BDI-based approaches and the implementation of robot program through Turing-complete languages (like C in our case), which do not have any notion of event. Their model of perceptions is mainly based upon a polling-based interaction, in the sense there is exclusively the possibility to read the current state of sensors ed act accordingly. This causes a huge addition of *if* statements concerning the likely values of active sensors. It is easy to notice the differences between the explorations in Chapter4 where C implementation requires besides new functions to accomplish new goals, a lot of tests positioned in a large number of lines throughout the code.

Another relevant concern stems from this issue: once we have identified the sort of tests we have to introduce, we wonder → *where must we locate them?* This choice turn out fondamental for reaching a well-engineered outcome,

because the lack of events means that there is the possibility to miss some relevant changes and so, to not handle all the likely situations -in the worst case, the hazardous ones. Thereby, another question grows out: *have they been put suitably within the code?*.

This is a tricky problem, and here's why choosing a right level of abstraction for engineering robot application, does not mean dying overwhelmed by complexity.

```
if(picked){
    if( !(force > -500 && force < -5) ){
      picked = false;
       /* stop the robot */
      wb_differential_wheels_set_speed(0, 0);
      printf("Cylinder has fallen. Anybody retrive it.");
      break;
    }
}
```

The snippet reported above is the one that checks whether the cylinder retrieved is fallen. This test -along with that which controls the battery level- is necessary in order to detect the occurrence of a critical situation, and must be placed in every relevant point of the program. The point is the meaning of the word "relevant". Let's suppose that each piece of code representing the fulfillment of a specific activity, takes x_i milliseconds to execute its operations, while sensor values are updated every y milliseconds. In order to not miss any significant condition of the whole world, we need to locate the above code in the right points inside the program, so that the controller is able to verify whether such condition is satified or not. Thus, that *if* statement should be performed hypothetically by z milliseconds, where
$z \leq y \leq \sum_i x_i$

```
#define TIME_STEP 40
****
static void pickUp(){
   checkForce();
   /*do something*/
   /*======*/
   return;
```

```
}

int main(){
   while (wb_robot_step(TIME_STEP)!=1) {

    checkForce();
/*do something*/
/******/
if(obstacleDetected){
       pickUp();
     }
     checkForce();
   /*do something*/
/******/
   wb_robot_step(TIME_STEP);
   }
}
```

As we can see, a growing complexity for implementing the controller program, has arisen from the addition of the ability to determine when the cylinder picked up is fallen. Thereby, this approach surely does not encourage programmers to enhance the robot behaviour with more complex skills and, moreover, does not promote the reuse of such code upon another hardware system.

"When X happens, do Y" is a basic statement for programming control agents. That *when* indicates a specific situation of the world that should have been detected whenever, inside the robot's behaviour. However, with standard languages this is not possible, therefore adding a set of tests to locate in right places of the source code, turns out necessary. But, in case there are a lot of events to handle, the growing addition of *if statement* becomes a sort of **pollution** of the agent's logic. This **pollution** complicates both the main activity and the different subtasks, and brings to affect the whole robot's behaviour as well.

The utilization of a **JaCa** -based approach overcomes this problem, since the agent has got plans which react in order to manage those events. Indeed, to execute all the operations before explained, in Jason agents it is necessary to insert only this plan

```
@fallCylinder[atomic] +fallen: picked(true) <-
                                .drop_all_intentions; .print("Cylinder has fallen.");
                                 stopAcquire; stop.
```

which is triggered immediately, when the perceptive layer notify it by means of a signal.

2 Modularity and Compositionality

Considering Chapter4 and what discussed so far, we got that programming robot by means of agent plans, allow to obtain a modular solution, so that its behaviour (agent-based controller) can be extended somehow, without totally changing any other part of the agent source code. That is, we can introduce in such way, a new expertise or skill by introducing slight modifications to robot behaviour, like adding new plans and / or beliefs. Hence, *the robot behaviour is augmented without the need of a strategy that entails to write the robot control program from the scratch.*

In our explorations indeed, we have a robot that earlier can avoid the encountered obstacles, but when we said "if you see a red cylinder, pick it up with your gripper!" what we have to add in order to make up a brand new robot which can retrieve red cylinders while avoids the environment obstacles?

As we saw in the Jason implementation we can define new subtasks -whose can be subdivided themselves- taking into account and managing the dependencies among them. In fact, looking throughout the implemented agents, several piece of code ar totally the same, like those plans an agent perform so as to avoid an obstacle

```
+!avoid(0) <- .print("Central obstacle"); backward; turnAround;
              .wait("+obstacle(false)"); -avoiding.
+!avoid(D): D < 0 & maxSpeed(M) <- setSpeeds(-M/2,M/2);
              .wait("+obstacle(false)"); -avoiding.
+!avoid(_): maxSpeed(M) <- setSpeeds(M/2,-M/2);
              .wait("+obstacle(false)"); -avoiding.
```

or those ones that define the basic random motion

```
+!moving <- while(true){
            .random(R); !move(R);
            }.
+!move(N): N <=0.5 <- forward; .random(R); W = R*10000; .wait(W).
+!move(N): N <= 0.75 & N > 0.5 <- turnLeft; .random(R); W = R*10000/2;
                                  .wait(W); forward; .wait(1000).
+!move(N): N < 1 & N > 0.75 <- turnRight; .random(R); W = R*10000/2;
                                  .wait(W); forward; .wait(1000).
-!move(_) <- println("move failed").
```

When the robot's abilities are growing, we had to act some changes or add similar plan -for the same triggering event- just because of the need to reach a certain level of coordination among the (sub)tasks. Whereas, using the C programming language, the addition of new features and skills involves sometimes several changes in different points of the agent source code.

These are the same for all the agents and even considering that the first we use to put our agents into practice, was different from the second (the former is defined by Webots owners, the latter is an actual commercial robot), the above plans are likewise used inside all the experiments. So we assume that, such modularity is useful even with regard to hardware aspects: in fact, we used agents with (some) identical plans for controlling different robot and this is one of the aim of this thesis, we mentioned in the introduction. Therefore with this approach we are able to make up autonomous system controllers whose can likely run upon several, different hardware -at least, with some slight changes in the middle layer.

This is one of the most significant features that an agent-based robot programming -especially BDI-based one- bring to programmers who are involved in robotics. Indeed, it turns out to be extremely useful having a bunch of well-known plans and beliefs whose aim to fulfill certain activities and accomplish some goals, and that a programmer can use for more than one specific robot application, and this is not possible -or at least it would be tough- by using classical programming languages.

However, the more skills and abilities the robot acquires, the more a Jason programmer has to cope with the coordination among the tasks whose aim to accomplish certain goals. This coordination encompasses: the deployment of beliefs whose deletion and addition will help to sort out the different activities, their straightforward positioning inside the plans already implemented and the definition of new suitable plans. In fact, looking at Chapter4 and

more in detalis in AppendixB, turns out clear that a good part of defined beliefs are useful to manage the overlapping among the activities (eg. avoiding, ~delivered and so on). Jason programmers besides, in order to provide a well-engineered outcome, must provide some features that are not so easy to ensure, such as find a mechanism for avoiding name clashes between goals and belief from separate modules[1] or control whether certain beliefs can be used by all the other tasks or not.

3 Performance Analysis

Even though issues concerning the performance that characterize BDI-based robotic control systems are not so relevant for the objectives of this thesis, we believe it is significant to discuss at least few aspects involved. So, in this section we want to give a brief consideration which turns out meaningful with regard to possible future refinements and enhancements -whose will present in the next chapter.

Enforcing a layered architecture to model a robotic control system, we ought to take into account the latency that such kind of architecture is likely to introduce, if we want to put into practice -onto a real robot- our work in the future. That is, according to these values we can understand whether put our agent program into a cognitive agent to control a real robot is feasible or not. The values we mentiones are simply some timestamps that help to calculate roughly the time elapsed between:

- when a given situation is detected by the sensors and the agent trigger the suitable plan (*reactive time*);
- when a particular action is executed by an agent and the time at which this action is actually applied from actuators (*response time*).

In many robotic applications, a robot needs to guarantee real-time properties only for a small subset of its tasks which are critical for safety reasons or the robot functionality, therefore the above delays should assume a value

[1] We could name the set of plans and beliefs related to a specific task to perform, a **module**.

as low as possible. Making few elementary tests[2] we can get these significant delays. Such values have been obtained by means of a notebook which works on a 2.13 GHz Intel®Centrino Core 2 Duo™processor, using Windows 7©64 bit operating system. The *reactive time* of the implemented system is around 2 milliseconds whereas the *response time* is around 40 milliseconds.

The second time is greater than the first just because of the set of previous operation that were carrying out by the robot, such as "stop the wheels" or "close the gripper".

However, such values does not tell us that the system implemented so far is well-suited also for real-time applications. There are too many aspects that must be considered, such as the type of hardware used -as mentioned before- or the simulator's characteristics. Though, the values measured are low enough to suggest us that the way paved is pretty correct. Of course a deeper analysis of the architecture should be attempted so as to bring our work onto actual autonomous systems whose have to take into account real-time aspects. So, we have not the certainty that our work will turn into a proper approach to develop real-time robotics applications, however we have been moving towards the right direction.

Summing up, on the one hand the BDI-based approach provide an high-level, modular method to program robotic control system and promote the reuse of a part of plans for developing new ones. On the other hand C programming language ensures more performance compared with Jason which is a Java-based language, so runs upon a JVM and it is well-known that Java is slower than C (up to 10 times slower in some benchmarks). Considering that, for hard real time applications, a **JaCa** implementation might not be suitable.

Generally speaking, because of the wide range of components that may be involved in a robot application, it is actually unfeasible to build a program where a reaction to each event is explicitly modelled. In fact, the number of parameters to take into account would be extremelly huge, and the development of that, in particular with C, would become very slow.

[2]Through basic timestamps we can verify when a low battery value is detected by the robot plaform and when the related plan is activated; and when a certain command is sent by the agent and then actually issued by the robot platform.

Chapter 6

Conclusions and future work

It is now time to summarize the work presented in the course of this thesis, pointing out the goals achieved, the contribution brought and its shortcomings, as well as the feasible future work.

The main goal of this thesis, was to think, model and then implement a purposeful and simple architecture to exploit an agent-oriented approach, like the BDI one, for robot programming.

The course of this thesis started presenting the basics of robotics, from the definition of robot to the adoptable architectures whose could control it and the existing programming platforms to implement these architectures. Afterwards, the agent oriented programming was presented, focussing on the BDI language Jason, that has been employed, along with CArtAgO, as the programming framework for the resulting system. The next step was concerning how to set up the actual software architecture derived from the **JaCa** framework nature and how to connect it with the Webots simulator in order to put into practice what explained previously.

Once the software simulation system was defined, a chapter dedicated to some experimentations was necessary so as to verify concretely the outcomes. These experiments encompass a good set of relevant problems for robot programming. The main objective was to analyse how the modularity and readability of robot programs could be improved by adopting an agent-oriented BDI-based programming model.

As we can see in the last chapter indeed, fragments of code which have

been created, refined and reused with the purpose to reduce the workload for the agent programmer when aims to extend the robot's skills. This has been achieved thanks to the **JaCa** nature which, unlike standard programming languages, allows to avoid that we called the *if pollution* of the agent's program.

That is, the contribution coming from this thesis is a useful method to exploit deeply, fruitfully and in a modular way the **JaCa** framework, for the first time, to program a robotic control system, so that it can be reused further in different applications and with different sort of robots.

Of course, we have not exhausted the space of all possible issues regarding all the possible aspects of robotics that we ought to take into account during the experimentations as well as the features required to achieve a perfect modularity. Thus, a good number of feasible extensions could be investigated in future projects, first of all additional requirements may be satisfied to improve agent-based robot programming:

- improving modularity and encapsulating of plans, in particular for integrating proactive and reactive agent behaviour;
- allowing for private beliefs, to plans / intentions, i.e beliefs used only in the context of those plans / intentions.

Another way that would be compelling to pursue will be devoted to extend the explorations towards multi-agents experimentations. Since our investigations entail a one-to-one mapping between a robot and an agent, it could be interesting to explore the single robot programming with more agents which control it. Such approach is useful in case the set of tasks the robot has to accomplish, are extremelly heavy. Furthermore, it is worth studying the aspects concerning the multi-robots systems, such as the communication, the coordination and the interaction among them -along with the problematics that will come up.

Besides, as we roughed out in Chapter 5 there is the need to pave the way towards a system that should guarantee bounded reaction and response time to events, since Jason does not give any certainty about the lapse of time that a given action takes to be completely carried out. As far as the future extensions are concerned, this is likely the first requirement that has

better to be accomplished. Indeed, in robotics and embedded systems, the real-time requirements satisfaction is maybe the most important factor, in order to set up a well-engineered robotic control application. But if we want to achieve this outcome, thinking again about the architecture is sensible. So, by looking up throughout the literature relevant works can be found according to possible extensions of the BDI architecture, for facing properly the real-time requirements. For instance, allowing for defining priorities of plans -with the same triggering event- which promotes non-determinism in choosing a plan to perform.

As remarked in the beginning of the thesis we aimed at thinking in perspective, even when a pre-programmed approach is not satisfactory anymore. For some type of tasks, it could be not feasible to design a priori the overall set of plans useful to achieve them. We may exploit the planning and learning techniques inside Jason libraries, so that plans can be added dynamically to the agent. So, the agent programming would be moving towards a situation in which there are different "holes" inside the robot's behaviour and knowledge, where it becomes necessary to have a **non-completely defined strategy** as if some robot's methodologies and abilities can be added or modified in a dynamic way, according to new expertise and experiences gained over time. That is, the journey we have started, encompassing the programming languages studied, will connect to the Artificial Intelligence domain, integrating techniques such as genetic algorithms or neural networks, to improve robots autonomy.

Bibliography

[1] D. Dennett
Intentional systems. Journal of Philosophy 68, 87–106, 1971

[2] M. Bratman
Intention, plans, and Pratical Reasoning. Harvard University Press, 1987

[3] R.H. Bordini, J.F. Hubner, M. Woodridge
Programming multi-agent systems in AgentSpeak using Jason. Wiley Series in Agent Technology, 2007

[4] A. Omicini
Multi Agent Systems course, Agents as Intentional Systems, 18-51, 2012.

[5] A. Omicini
Multi Agent Systems course, Agents & Artifacts: Definitions & Conteptual Models, 40-42, 2012.

[6] R.S. Amant, A.B. Wood
Tool use for autonomous agents. AAAI/IAAI 2005 Conference, 9-13. Pittsburgh, 2005

[7] Maja J. Matric
The Robotics Primer, 2007

[8] B. Gates
A robot in every home. Scientific American, 2007

[9] D. Norman
Cognitive artifacts. Designing interaction: Psychology at the hu-

man–computer interface. Cambridge University Press, 17–38. New York, 1991

[10] G. Biggs, B. MacDonald
A survey of Robot Programming Systems. Proceedings of the Australasian Conference on Robotics and Automation. Brisbane, Australia 2003

[11] Oxford Dictionaries.
Retrieved 4 November 2012.

[12] J.L. Jones, D. Roth
Robot Programming. A pratical guide to Behaviour-Based Robotics. McGraw-Hill, 2004

[13] RobotC official web site: `http://www.robotc.net/`

[14] Microsoft Robotics Development Studio web site: `http://www.microsoft.com/robotics`

[15] O. Michel
Webots: Professional Mobile Robot Simulation. Journal of Advanced Robotics Systems, 39-42, 2004

[16] UrbiForge: `http://www.urbiforge.org/`

[17] `http://en.wikipedia.org/wiki/Workcell`

[18] R. Bordini, L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, A. Ricci
A survey of programming languages and platforms for multi-agent systems. Informatica 30, 33-44, 2006

[19] A. Ricci, A. Santi
CArtAgO by examples, Version 2.0.1. `http://cartago.sourceforge.net/?page_id=47`

[20] A. Santi, M. Guidi, A. Ricci *JaCa-Android: An Agent-based Platform for Building Smart Mobile Applications. Proceedings of Languages, methodologies and Development tools for multi-agent systems (LADS), 2010.*

[21] AS. Rao
AgentSpeak(L): BDI agents speak out in a logical computable language. Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World, 22-25. Eindovhen, January 1996

[22] N. Madden, B. Logan
Modularity and compositionality in Jason. In Proceedings of Int. Workshop Programming Multi-Agent Systems. ProMAS 2009.

[23] P. Ziafati, M. Dastani, J.J Meyer, L van der Torre
Agent Programming Languages Requirements for Programming Cognitive Robots. Promas 2012.

[24] M. Wooldridgw, NR. Jennings
Intelligent agents: theory and practice. The knowledge engineering review 10(2), 115-152, 1995

[25] C. Wei, K.V. Hindriks
An Agent-Based Cognitive Robot Architecture.

[26] A. Mordenti
Artefatti di coordinazione per agenti in Smart Environment. July 2010.

[27] A. Ricci
Environment programming in Multi-Agent Systems. WOA Mini-Scuola, 2011.

[28] A. Ricci, M. Viroli, A. Omicini
The A&A Programming Model and Technology for Developing Agent Environments in MAS. Programming multi-agent systems, 4908, 91–109, 2007

[29] A. Ricci, M. Piunti, M. Viroli *Environment programming in multi-agent systems: an artifact-based perspective. Autonomous Agent Multi-Agent Systems, 23, 158–192, 2011*

[30] A. Omicini, E. Denti
From tuple spaces to tuple centres. Science of Computer Programming, 41(3),277–294, Nov. 2001

Ackwnoledgments

With this thesis I fulfill a fascinating, and also hard journey, which have been engaging me along the last five years. There were not just easy and lightweight periods, sometimes I felt melancholy, tired and unconfident. This is the reason why the people who have been next to me so far, deserve their own chapter in this paper.

A special thanks goes to my supervisor Alessandro Ricci, who gave me the opportunity to study and work on a topic which has always attracted and intrigued me. He has beared my frequently doubts, oversights and blunders in a extremely stimulating way, prompting me to work with more and more self-denial.

I am deeply greatful to my family, for all the fundamental suggestions and the continuous support, allowing me to study without the burden of time or money. They have encouraged me to undertake this long journey without any doubts about my abilities and my will. Surely I would never been who I am today without them.

Afterwards, a sincere thanks goes to all my friends who have been with me during this years. The weekends, the jokes, the holidays, the talks, the drunks, the hangover with them are simply unforgettable. Some of them were simple schoolmates or teammates, then they have turned into something more important for me. I am also truly greatful to my girlfriend who always make me feel special and try to improve my self-esteem (much of work is left to do). I did not want to point out who those people are, because the ones who care about me, know that I am thanking them.

I would have liked to devote this people more than a mere black and

white page inside a dissertation, so I hope I will give them the same special satisfaction and emotions they convey me. All my efforts are dedicated to them, as well as my grandpa who is always close to me, (even though not physically) supporting my day-to-day life, I miss you.

Appendix A

SensorInfo

```
public final class SensorInfo implements Serializable{

private  int id;
private  Double sensorValue = 0.0;
private int[] intValues = new int[]{};
private Double x = 0.0;
private Double y = 0.0 ;
private Double z = 0.0;
private Double[] doubleValues = new Double[]{};

public SensorInfo(int id, double value){
    this.id = id;
    sensorValue = value;
}
public SensorInfo(int id, int[] image){
    this.id = id;
    intValues = image;
}
public SensorInfo(int id, Double x,Double y,Double z){
    this.id = id;
    this.x = x;
    this.y = y;
this.z = z;
}
public SensorInfo(int id, Double[] values){
    this.doubleValues = values;
}
```

```
public double[] getCoord(){
    return new double[]{x,y,z};
}
public void setVal(Double value){
    sensorValue = value;
}
public void setRGB(int[] components){
    this.intValues = components;
}
public double getVal(){
    return sensorValue;
}
public Double[] getDoubleValues(){
    return this.doubleValues;
}
public int[] getIntValues(){
    return this.intValues;
}
public int getId(){
    return id;
}
}
```

Appendix B

Task Suspend/Resume Agent

```
/* Initial beliefs and rules */

maxSpeed(100).
storagePoint(0.41,-0.41).
charger(-0.38,0.38).
threshold(0.04).
reachCylinder(false).

/* Initial goals */
!start.

/* Plans */

+!start <- !init(Id); focus(Id); println("focus achieved");  accept; !waitConnection.

/*Plans whose take care of the inital phase of the agent system*/

+!init(Id) <-  makeArtifact("middleware","artifacts.Middleware",[4444],Id);
 println("Artifact created").
-!init(_) <- println("artifact creation error"); .stopMAS.

+!waitConnection: connected(true)
<- deltaBased(false); ?storagePoint(X,Z); +targetPoint(X,Z);
setUpTargetPoint(X,0,Z); +reachCylinder(false);
 ?threshold(T); setThreshold(T); acquireData; !moving.
+!waitConnection <- .wait(500); !waitConnection.

/*The above eight plans perform the operations which allow the
* robot to reach both a lateral and a central cylinder*/

+cylinderLeft: reachCylinder(false) & not insideArea & not charging
<- -+reachCylinder(true); stop;
```

```
.print("Left Cylinder");
 setSpeeds(0,10); !reach.
+cylinderRight: reachCylinder(false) & not insideArea & not charging
<- -+reachCylinder(true); stop;
.print("Right Cylinder");
setSpeeds(10,0); !reach.
+cylinderBeyond: reachCylinder(true) & maxSpeed(M)
<- .print("cylinder beyond"); turnAround;
.wait(20000); setSpeeds(M/5,M/5); !reach.

+!reach: charging <- .wait({-charging}); .print("Resume cylinder reaching"); !reach.
+!reach: reachCylinder(true) <- .wait("+cylinderBeyond",5000).
+!reach: reachCylinder(false) <- .wait("+cylinderDetected",3000).
-!reach <- .print("Reach lateral cylinder failed."); -+reachCylinder(false).

+cylinderDetected: not picking & not delivering & not charging
<- +picking; -+reachCylinder(false); stop;
.print("Cylinder detected"); catching; !block.


/*Plans useful to avoid obstacles*/

+obstacle(true): (delivering | charging) & delta(D)
<- .drop_intention(balance); +avoiding;
stop; !avoid(D); forward; !computeGap;
!balance; forward.
+obstacle(true): delta(D) & not avoiding & not picking & not reachCylinder(true)
<- .print("Obstacle: ",D); +avoiding; stop; .wait(50); !avoid(D).

+lateralObstacle("left") <- println("high left"); !avoid(-1).
+lateralObstacle("right") <- println("high right"); !avoid(1).

/*The plan hereunder carries out all the necessary operations
* to enforce, when a well-known point is reached while the delivering
*task is running*/

+targetReached: delivering & not charging
<-  .drop_desire(balance); .drop_desire(computeGap); .drop_desire(avoid);
stopAcquireDistance; -delivering; .wait(100);
.print("TARGET POINT REACHED"); stop; .wait(40);
putDown; -desired(_); ?orientation(O); D2=O+180; +desired(D2);
!balance; .print("balanced");
forward; waitExit; .wait("+~around"); -~delivered.

+!raiseRay: threshold(T) <- T2=T+0.03; setThreshold(T2).

/*These plans help to satisfy safety aspects concerning the
*cylinder delivering*/
```

```
+outOfStorageArea <- println("Out of target area");
restartAcquireDistance;
         !raiseRay; -insideArea; +~around.

@noAccess1[atomic]+noAccess: not delivering & ~around
<- +gettingOut;
.print("Too close to storage area!! Go away!");
stop; .wait(40); turnAround;
.wait(2000); forward; .wait(1500).
+noAccess: delivering & firstTime & not charging
<- -firstTime; +insideArea;
println("Almost reached"); stopAcquireDistance.


/*Standard plans to achieve the desired heading */

+!computeGap: location(X,Y,Z) & targetPoint(Tx,Tz)
<- computeAngle(X,Z,Tx,Tz,Desired);
-+desired(Desired).

+!balance: desired(D) & orientation(O) & O > D-0.6 & O < D+0.6
<- stop; -~around; +firstTime; forward.
+!balance: desired(D) & orientation(O) & ((O-D >= -180 & O-D < 0) | (O-D >= 180))
<- setSpeeds(25,-10); .wait(20); !balance.
+!balance: desired(D) & orientation(O) & (O-D < 180 & O-D > 0 | O-D < -180)
 <- setSpeeds(-10,25); .wait(20); !balance.
+!balance <- .wait(20); !balance.

/*This bunch of plans perform the random navigation and are suspended
* when another more relevant task is triggered*/

+!moving <- while(true){
.random(R); !move(R);
}.
+!move(N): charging
<- .print("wait recharging");
.wait("-charging"); .print("after recharging");
!move(N).
+!move(N): delivering
<- .print("Wait delivering");
.wait("-~delivered"); .print("Delivered!");
!move(N).
+!move(N): avoiding
<- .print("Wait avoiding");
.wait("-avoiding"); !move(N).
+!move(N): reachCylinder(true)
<- .print("wait lateral cyl");
.wait("+reachCylinder(false)"); !move(N).
+!move(N): picking <- .print("wait picking"); .wait("-picking");
```

```
.wait(50); !move(N).
+!move(N): N <=0.5
<- forward; .random(R); W = R*10000; .wait(W).
+!move(N): N <= 0.75 & N > 0.5
<- turnLeft; .random(R); W = R*10000/2;
.wait(W); forward; .wait(1000).
+!move(N): N < 1 & N > 0.75
<- turnRight; .random(R); W = R*10000/2;
.wait(W); forward; .wait(1000).
-!move(_) <- println("move failed").

/*These three blocks of plans concerning the whole
* cylinder picking operation*/
+!block: charging
<- .print("Blocking suspend, wait recharging.");
open_grip; arm_up; .wait({-charging});
.print("Blocking resumed"); !block.
+!block <- .wait("+caught(true)",5000); !pick.
-!block <- println("block failed"); !pick.

+!pick: charging
<- .print("Picking suspend, wait recharging.");
.wait({-charging}); .print("Picking resumed");
!pick.
+!pick: caught(true)
<- arm_up; .wait("+picked(true)",3500);
!go_ahead.
+!pick <- .print("Cylinder not found"); arm_up; turnAround;
.wait(2000); -picking.
-!pick <- .print("pick failed"); !go_ahead.

+!go_ahead: charging
<-.print("Delivering suspend, wait recharging.");
.wait({-charging}); .print("Delivering resumed");
!go_ahead.
+!go_ahead: picked(true)
<- println("Cylinder picked up"); -picking;
+~delivered; +delivering; !computeGap; !balance; forward.
+!go_ahead <- println("Cylinder picking failed"); stop.

/*Basic plans to avoid obstacles*/
+!avoid(0) <-  backward; turnAround;
.wait("+obstacle(false)"); -avoiding.
+!avoid(D): D > 0 & maxSpeed(M)
 <- backward; setSpeeds(M/2,-M/2);
.wait("+obstacle(false)"); -avoiding.
+!avoid(_): maxSpeed(M)
<- backward; setSpeeds(-M/2,M/2);
.wait("+obstacle(false)"); -avoiding.
```

```
/*Atomic plan, that stop completely robot's execution, since
* the picked cylinder has fallen*/

@fallCylinder[atomic] +fallen: picked(true)
<-.drop_all_desires;
.print("Cylinder has fallen. Anybody retrieve it");
stopAcquire; stop.

/*Plan which reacts to the (critical) situation, when the battery
* is getting exhausted */

+lowBattery: location(X,Y,Z) & charger(Cx,Cz) & not charging & orientation(O)
<- mark; +charging; ?battery(B); .print("Battery low! Level: ",B);
stop; -+targetPoint(Cx,Cz); setUpTargetPoint(Cx,0,Cz);
!computeGap; !!balance; .wait({+targetReached}); stop;
.wait(3000); .print("RECHARGED"); ?battery(B2);
.print("Current battery level: ",B2); -+targetPoint(X,Z);
setUpTargetPoint(X,Y,Z); !computeGap; !!balance;
.wait({+targetReached}); stop; .print("Previous pt reached");
?storagePoint(Tx,Tz); -+targetPoint(Tx,Tz);
setUpTargetPoint(Tx,0,Tz);
!restoreOrientation(O); -charging.

-charging: delivering <- .print("Restart delivering");
!computeGap; !balance.

@restore[atomic]+!restoreOrientation(D)
<- .print("Restoring"); -+desired(D); !balance; stop;
  .drop_intention(balance); .print("Restored"); .wait(500).
```