Master of Science Thesis

ROSAPL: towards a heterogeneous Multi-Robot system and Human interaction framework

Emili Boronat Roselló

Advisor: Dr. Javier Vázquez-Salceda

September 2014
"To my beloved father
May the force be with you"
Abstract

The appearance of numerous robotic frameworks and middleware has provided researchers with reliable hardware and software units avoiding the need of developing ad-hoc platforms and focus their work on how improve the robots’ high-level capabilities and behaviours. Despite this none of these are facilitating frameworks considering social capabilities as a factor in robots design.

In a world that everyday seems more and more connected, with the slow but steady advance of the Internet of Things to many aspects of our daily lifes, the lack of social capabilities in a robot limits developers and researchers on areas where robots are seen as part of a solution, and not the solution. This thesis states that a social layer should be accessible in any robotic platform in order to ease the development of systems where such platforms are just a piece in the whole socio-technical system. As result of this effort we present the ROSAPL framework to develop social robots on top of ROS middleware.

We tested our approach in a real scenario at IBEC’s Robotics group in the context of the InHANDS, which project tries to assist a handicapped persons in the kitchen. For them we designed and implemented a prototype to proof ROSAPL applicability. This latter will be fully implemented to offer real functionalities for the kitchen.
Acknowledgements

I would like, first, to thank my Mother and Brother. They have been there is such dark times, backing me up, encouraging me and letting me keep going on with the Master and, what seemed impossible some times, to finish this Thesis. This, and whatever comes next it is for both of you!

Second, to Javier Vázquez, who has been all patience with me all this time. Your advices where invaluable to keep me focused. Thank you for all your unconditional effort, receiving me at your home and helping me to improve my writing skill until the last minute. Also to Karina Gibert, who guided and supported me along the whole Master, helping me to go through all sort of bureaucratic experiences. It was a pleasure to meet you and your bright personality. Finally to all the teachers of the Master: Ramon, Pablo, Ulises, LLuís, Miquel, Joan and many others who lend us, students, your knowledge and guidance.

Third but not less important, to my beloved Maria, who has been there supporting me all this whole year, determined to not let me die of starvation while I was working and studying at nights; gently waking me up every single day with her smile.

Daria, thanks for your unconditional patience listening and your Grammarnazi review obsession. That saved Javier some dioptres. You know, the tea you brought was the fuel that kept me up most of the nights. Fernando, our discussions where enlightening, thank you for being there all the time, you have done your part in this work too. I would not want to left anybody out here, there is many people I encountered in the Master that helped me learn everything I know today, Thank to all of you too.

I would like to finish this section giving thanks to Joan Aranda and Alicia Casals, who offered me a place at IBEC, that, literally, saved my life. I would like to extend it also to all my fellows there, Olga, Vijay, Xavi and Manu, for their patience and support with the new guy.
# Contents

List of Figures viii  
List of Tables ix  

1 Introduction 1  
   1.1 The problem at hand 2  
   1.2 Application of AI and MAS 3  
   1.3 Structure of this thesis 4  

2 InHANDS project 7  
   2.1 Project Description 8  
   2.2 The Laboratory 10  
   2.3 Current Architecture 11  
   2.4 Scope of this thesis within InHANDS 16  
   2.5 Domain specifications 17  
   2.6 Design Requirements 18  
   2.7 Proposed Approach 19  

3 State of the Art 21  
   3.1 Assistive Technologies 21  
   3.2 Multi-Agent Systems 23  
   3.3 BDI programming 25  
   3.4 ROS: Robotic Framework and Middleware 28  

4 The ROSAPL Framework 33  
   4.1 Framework Bases 34  
   4.2 Framework Concepts and Models 36  
   4.3 Overview of the ROSAPL workflow 45  
   4.4 Architecture 48  
   4.5 Development Cycle 55  

5 InHANDS powered by ROSAPL 59  
   5.1 Prometheus Overview 60  
   5.2 InHANDS System Specification 63  

vi
List of Figures

2.1 InHANDS scene sketch ............................................ 9
2.2 Photography of the lab ........................................... 10
2.3 System Modules .................................................. 12
2.4 scene perception .................................................... 13
2.5 User interface overview .......................................... 15
2.6 Scope of this thesis within InHANDS ......................... 16
3.1 Overview of the BDI architecture ......................... 26
3.2 2APL deliberation cycle ......................................... 27
4.1 ROSAPL Agent programming layers ....................... 37
4.2 Example of plan in the proposed representation .......... 42
4.3 Example of capabilities implemented using relations, as shown in [Nun14] ......................................................... 44
4.4 ROSAPL approach workflow ..................................... 46
4.5 ROSAPL agent internal architecture ....................... 50
4.6 Proposed archetypical module hierarchy and relations for a robotic agent .................................................. 51
4.7 Multi-robot architecture ......................................... 52
4.8 Overview of possible human-robot interaction system as a robot interface ................................................. 54
4.9 Overview of a possible assistance system based on agents ............... 55
5.1 Detail of the modules inside the design scope ........... 60
5.2 The Phases of Prometheus methodology .................. 61
5.3 InHANDS Goal Overview Diagram ......................... 64
5.4 InHANDS System Overview Diagram ....................... 65
5.5 Data Coupling Diagram ......................................... 66
5.6 Agent Role Grouping Diagram ................................ 67
5.7 Avatar Design Diagram ........................................... 68
5.8 Robot Design Diagram ............................................ 68
5.9 Scene Agent Diagram ............................................. 69
5.10 Plan Generation Capability ..................................... 71
5.11 Plan Monitoring Capability ..................................... 72
List of Tables

2.1 Reasoning Module inputs and outputs .................................. 17
2.2 Functional Requirements ................................................. 19
2.3 Non functional Requirements ............................................. 20
4.1 Types of task relations ..................................................... 42
4.2 Node Types ............................................................... 43
5.1 InHANDS Percepts and Actions ........................................... 64
5.2 InHANDS Data Sources .................................................... 65
Chapter 1

Introduction

Nowadays there are dozens of robotic platforms under development, ranging from mobile platforms to full anthropomorphic robots, from home use Roomba vacuum cleaner [TD07], research platforms such PR2 [BRJ+11] or industrial solutions such as baxter [Fit13]. However, the development of a reliable robot requires the combined effort of a team of specialist in many fields to build and connect all the hardware and software modules that compose a robot.

In order to ease robot development numerous robotic frameworks, middlewares and platforms have emerged to provide developers with reliable hardware and software units they can use and extend. This has allowed research groups to avoid the need of developing ad-hoc platforms and focus their work on how improve high level robots capabilities.

However these platforms and frameworks have a lack of social capability as, right now, none of them will know how to socialize out of the box with other units. This limit the work some one can carry out on those platforms. Specially if the research field is robot-robot or human-robot interaction. Most of the research done on cooperative robotics such as Swarm robotics, team building, cooperative task allocation and cooperative execution are build upon custom ad-hoc platforms. Therefore, if someone is starting to dive into these fields normally will require to build the social layer from scratch.

A few years ago there was a similar difficulty in how to control and configure the hardware a robot has. Robotic middlewares such ROS[QCG+09], YARP[MFN06] or OROCOS[Bru01] help robotics development by unifying efforts to solve most of the common issues developing robotic platforms and managing the hardware related to it. That was possible thanks to the socialization of successful solutions and reusable modules. And the factor of success was that all these modules could be tested by the whole community of developers. Despite the success of this approach, it is mainly applied on building a unique robot system. Not only a type, but a single instance. So it is very
easy to found modules that will help developing a new robotic platform on your own, yet most of the times having different units of the same platforms working together requires a significant amount of work.

1.1 The problem at hand

Although some of the robotic platforms are now including some low-level robot-to-robot communication, there is not a coordinate effort. To prevent the rising of a robotics *Tower of Babel* the main objective of this Master Thesis is to provide an agent-oriented solution to develop multi-robot systems. In order to support an agent-oriented design, a new communications framework has been developed where the provided design could be build on.

Multi-Agent Systems (MAS) is an area with 2 decades of research on software socialization, proactive behaviour, social organization structures, reasoning and modularization, all these of interest for cooperative robotics in both, research and commercial applications. One of our claims in this work is that robot developers need to have a framework that not only helps developing a robot platform and its individual capabilities but also its socialization capabilities, otherwise product development cycles to achieve such social behaviours will be too long for commercial products.

This thesis also embraces the task of designing an intelligent system that should be able combine different robotic platforms and their capabilities in order to assist a human user on its activities in a specific scenario.

The specific use case this thesis will work with is actually a real design scenario. The *Institut of Bioengineering of Catalonia*\(^1\) (IBEC) has a laboratory that simulates a real kitchen where its robotic research group, lead by Alicia Casals\(^2\), conducts research in new Assistive Technologies (AT) based in robotics and natural interfaces. The current hosted project, Interactive robotics for Human Assistance iN Domestic Scenarios (InHANDS), tries to achieve a system that will integre several robotic platforms in a domestic environment in order to assist handicapped or elder people in their daily live. The kitchen, as remarkable elements, counts with 3 robotic platforms and a immersive graphical interface that projects the information directly in the scene. The robots are being programmed to be capable of manipulating the elements of the scene, execute some common task for the user and cooperate with him in some others, such as carrying heavy objects. The ultimate result is to have a human-robot interaction system that should be abstracted from hardware specifics, should be modular and scalable in number of capabilities, devices and scenarios. After presenting to the Robotics group the approach of

\(^1\)http://www.ibecbarcelona.eu
\(^2\)http://www.ibecbarcelona.eu/robotics
Since the main problem is related to how they manage the user representation, the knowledge and coordination of a heterogeneous group of robots a MAS approach seems fairly reasonable. Besides final commitment is to solve particular design case their requirements also encourages the idea that a basic multi-robot framework is necessary. In fact the case at hand could be generalized to other scenarios as, in essence, it is a system that will take into account artificial actors such as software modules and robots that are aware of their surrounding world to interact with humans in order to provide them with proper assistance doing their tasks. This could be the case also for other areas such as search and rescue, space exploration or structure construction, to mention a few.

In order to provide an implementable design it is necessary to formalize the main concepts and interactions such a system should have, focusing into satisfy InHANDS requirements. Next to that we will revise the current technologies available to address the requirements of the project not achievable by the current solutions used by the project. We will then adapt those technologies so they will be suitable to solve the design problem at hand.

The scope of this thesis is therefore limited to the needs of InHANDS project and will not try to attempt to achieve a full general framework, but rather a fairly simple one upon to build or inspire functional solutions with room for future improvements or enhancements.

1.2 Application of AI and MAS

The overall objective is to design the core of InHANDS that will perform several tasks that normally are considered AI problems such as planning, task allocation, cooperative and collaborative work, reasoning under uncertainty and so on. So it is not a surprise that the proposed solution to the problems comes from applying different AI techniques.

The MAS field specializes on how to make software sociable as well as proactive, enabling them to interact in order to achieve complex goals through combination capabilities at hand.

It is important to note that the solution of the present scenario implies the integration of a lot of different techniques and the fitting of these as well as its proper performance. Each one of them could represent a Master Thesis work for itself so we will try to keep things simple and apply general approaches to each of them focusing our main work in the multi-agent field in other to build a proper base upon later each particular subproblem could be addressed in a proper way. Wherever this will imply the addition of new agents or the
improvement of the capabilities the final ones will have.

The main contribution of this master thesis will consist on the exploration of multi-robot systems design and the proposal of guidelines and methodologies to achieve that. It will culminate with the proposal of a solution to a real scenario proposed by InHANDS project using them.

1.3 Structure of this thesis

The rest of this document is organized in 5 chapters and 2 appendices.

In chapter 2 we will introduce the IBEC’s lab and the InHANDS project. An overview of the equipment and technologies of the lab will be presented. We will also comment the current research done towards human activity recognition and how it relates to the project. Finally the scenario requirements are introduced to narrow down the scope of this thesis within the project.

In chapter 3 we present current technologies and solutions to the three areas this thesis is based on. We will describe what Assistive Technologies (AT) are and revise similar scenarios on this area that could present precedents for the project. We briefly introduce Multi-agent Systems (MAS) and related concepts and describe the platform that will be used for prototyping and implementation purposes: A Practical Agent Programming Language\(^3\) (2APL). The chapter finishes with Section 3.4 dedicated to the Robotic Operating System (ROS) to give a brief overview of its main concepts and functionalities in order to understand later descriptions in this thesis.

In Chapter 4 we expose strengths and weakness of the available technologies and which features ROS and 2APL cover the robotics development. We also revise where additional work should be done these two frameworks in a solution that will be enough to address general multi-robot systems design and implementation. The main concepts that this framework uses and builds on are presented along some examples in Section 4.2. We propose a basic architecture to get started in order to face complex multi-robot scenarios. At the end of the Chapter we discuss a suggested development cycle and which tools are available to develop multi-robot systems following the ROSAPL framework.

In Chapter 5 we give a description of the final design proposed to address InHANDS requirements. It is included also the descriptions of the methodology that has been followed as well as the decisions made to solve the challenges and issues encountered along the way to define the system goals and functionalities. We also present the description of the types of agents and its main capabilities as well the principal interaction protocols.

\(^3\)http://apapl.sourceforge.net/
1.3. STRUCTURE OF THIS THESIS

Finally in Chapter 6 we discuss about the provided approach and its potential generalization. We also revise if the proposed solution meets the InHANDS requirements and expose the final evaluation of it by the project team. Finally we propose future work lines based on ROSAPL to achieve a truly functional framework.

Two appendices are included in this document. Appendix A contains a basic user’s guide for the ROSAPL package. Appendix B includes the full proposed system design that will be used in InHANDS to implement a solution to its scenario.
Chapter 2

InHANDS project

The Institute for Bioengineering of Catalonia (IBEC) is an interdisciplinary research centre focused on bioengineering and nanomedicine, based in Barcelona. The institute currently has 16 research groups and 250 researchers and staff from 20 different countries. IBEC’s groups and their activities are organised into six research programmes:

- Cellular Biotechnology
- Biomechanics and cellular biophysics
- Nanobiotechnology
- Biomaterials, implants and tissue engineering
- Medical signals and instrumentation
- Robotics and biomedical imaging

The Robotics group’s research activity is mainly application-driven as new social and medical demands propose new scientific and technological problems, which become the research targets. The steering subjects, apart from the scientific interest that poses research challenges, are social responsibility and support for surgeons, rehabilitation and assistance staff. Designing and developing intelligent robotic strategies and control systems to assist people with disabilities, as well as medical personnel, implies the interpretation of human intention and context awareness.

The InHANDS project [AVM10] explores how novel human-machine interfaces can improve the user experience in Assistive Robotics environments.

In this Chapter we first present the InHANDS project in more detail. Section 2.2 describes the robotics Laboratory and its main elements. In Section
2.3 the current architecture of the platform is discussed. Section 2.4 determines the scope of this thesis inside the project. Section 2.5 describes the specifications and Section 2.6 requirements that will affect this thesis and in Section 2.7 we propose to approach the design scenario using agent oriented technologies.

2.1 Project Description

The InHANDS project is the continuation of the CAPDI project [APXC99], which set the basis of the current scene environment and integrated the first robot of the project. The current project iteration [VAC14], the second in InHANDS, focuses in human environment interaction. They explore modern and minimal intrusive ways on how people could interact with new generations of technology for assistance and care giving. Figure 2.1 shows a sketch of a test scenario. Their main purpose is the study, development and implementation of a robotic hyper-flexible cell in the context of a domestic environment to assist impaired or elder people.

The main idea is to set the basis to move a step forward on these technologies. This focus on elder or handicapped people does not exclude that the expected results of the project will be generally applicable. Thus activity monitoring will not be exclusive for elder people, but also could be used in children care or security systems together with anomaly detection algorithms. The main lines of research of the project are focused on visual perception of the environment and human action recognition and interpretation. In those lines a visual perception system is being developed to acquire the position and gestures of the user that is interacting with the kitchen. Once captured, in a second stage, this information will be processed to recognize and interpret the human actions from among a list of action patterns. In the end, all this information should be used to perform activity recognition to predict the main activity of the user through its actions.

The technical contributions of the project will be a modular platform with a multi-modal user interface designed in a way that could easily integrate new hardware to adapt it to any kind of needs user could have. This resulting platform should not only be seen as domestic assistance environment. Labour inclusion of disabled people is an issue in our society. In the mid and long term the InHANDS system should be able to be extended to other environments such as offices or industrial plants.

The main goal of the current iteration is to set the basic technological platform that will allow to demonstrate the concept. Once that will be accomplished it will be easier to improve one step at a time until a complete and functional smart kitchen environment is achieved. Future iterations will
2.1. PROJECT DESCRIPTION

concentrate on imply medical studies to direct the next steps in the project and to decide which functionalities should be improved towards state of the art solutions in the group’s fields of interest such as Assistive Robotics or Rehabilitation Therapies. However, this does not exclude the possibility to offer the platform to collaborate with researchers or companies interested in natural language processing, human-robot interfaces, healthcare monitoring or other related topics.

The platform is now under intense development. In particular, a significant effort has been made to update its core modules to use the Robot Operating System (ROS) as a middleware in order to standardize the platform. ROS is a meta operating system that provides an easy communication protocol to communicate different processes within and between computers and is specially designed to work with hardware drivers and robotic applications. This update is done also to easily accommodate a couple new robots (presented in Section 2.2).

1Section ?? of Chapter 3 provides a more detailed explanation about what is ROS and the most basic concepts about it or visit http://www.ros.org
2.2 The Laboratory

For its research the project has a laboratory that simulates a kitchen (see Figure 2.2). It has a couple of hobs and a sink. Its available space is around 18m$^3$. In it we can find 3 different robots, speakers, microphones and speakers as well as many cameras.

The perception is done through a set of cameras and microphones, the latter are essentially integrated in the rgb-d cameras that are used. The images and point clouds produced by the RGB-d cameras are used for object detection, recognition and identification, user tracking, gesture recognition, activity recognition and 3D scene reconstruction. There is room for adding more devices thanks to ROS infrastructure. Also it allows to access existent hardware from new software modules or packages without any need to modify the ones already using it.

The system gives feedback to the user through an immersive visual interface. This is generated on the kitchen wall and workspace using a short-range projector.

The three robots in the lab allow to apply actions to change the actual state of the world.

Figure 2.2: Photography of the lab
2.3. CURRENT ARCHITECTURE

Capdi
This is an old robot model, the first of its kind, that has a very big workspace volume but has very low manipulation capabilities. Thus this robot is mainly used for transportation and storage of goods. Has little to none direct interaction with the user.

Mico
This is a small weight, precise robot designed for the assistance of people with mobility disabilities. It is specially recommended to be mounted in a wheelchair and its anatomical structure is very suitable to interact with a person, e.g., example aiding them to drink or eat, move objects and manipulate them. It is used in InHANDS it to help users to manipulate objects in the working area.

Baxter
This is a new generation of industrial robots envisioned to work hand by hand with humans. The Robotics group at IBEC is exploring the possibilities of coordinating this type of robot with humans to carry along daily task like cooking. The principal use of this robot in the scenario to perform an independent task that requires 2 hands to help the user. It should be the robot capable of special tasks such as stirring or cooperative transportation.

Possible upcoming elements could be intelligent furniture ranging from automated cupboards to complete automate storage containers.

2.3 Current Architecture

The current implementation of the project is based on 4 modules, depicted in Figure 2.3. It follows the traditional sense-reason-act cycle in a centralized architecture where the fourth module is in charge of both user and environment interfaces. Commanded by these modules there are a variety of hardware devices and a robotic arm CAPDI and is used to move objects around to avoid users in wheelchairs to move around the kitchen.

The next subsections describe each module and its associated hardware in more detail. Explaining the capabilities and functionalities that will provide when all upgrades will finish. All the packages and modules are being or have been already ported to ROS and take advantage from standard implementations of robot capabilities such as motion planing, object recognition, robots perceptions and control algorithms.
CHAPTER 2. INHANDS PROJECT

Perception Module

The Perception Module, shown in the left of Figure 2.3, uses RGB and RGB-D cameras in order to maintain a geometric representation for the state of the environment, a semantic representation of what is in the scene and the whereabouts of the user inside the environment. The main processes of this modules are the Object Detector and the Motion Recognizer.

The Object Detector is incorporating object recognition and localization algorithms that notifies Perception Module about which and where are the manipulable objects in the scene. The submodule maintains an historic of the objects that has been around the scene, so its easy to find where they come from. It also help to identify similar items. The final ROS version of this modules will allow to access to object information such as object utility value to a given action, its shape and grasping points for each robot.

The Motion Recognizer module is mainly used to detect users in the scene, extract the key features and necessary data to feed the Activity Recognizer in the Reasoning Module. It also shares the motion detection information with robots so they could apply security policies wherever they have to.

The Perception Module also keeps an occupancy map to allow robot motion planning and monitoring in parallel to the other two functionalities. A graphical presentation of the output of this module is shown in figure 2.4. The scene is rendered for the information perceived by CAPDI, the arm hanging from the roof that can be seen in the image, together with the occupancy map and the known objects. The wall, the table and the shelf.

Figure 2.3: System Modules


2.3. CURRENT ARCHITECTURE

The current implementation of the Reasoning Module is based on a state machine and a set of conditions that trigger tasks such as pick an object mainly directly connected to the user interface input. So far this is the less developed module and the one requiring more effort to truly provide InHANDS with smart behavior. The challenges are many, but this iteration concentrates in Human Action Recognition and assistance plan generation from user activity in real time.

The reasoning modules provide, as basic functionality, the back end to the User Interface. Allowing the user to request the execution of any action. The main components of this module are the Activity Recognizer and the Task Planner, shown in Figure 2.3. The Activity Recognizer is the component where the group is putting more research efforts. There are two ongoing PhD Thesis conducted by Manuel Vinagre and Olga Mur focused on human activity recognition from spatial data. Manuel Thesis is working at general level trying to identify in real time which action user is performing from features extracted from 3D images provided by rgb-d cameras. Olga is focussing in activity recognition at fine motor skill level, tracking user hands to determine the user actions. The output of this module is the most probable action that a user is
performing. The list of considered actions is introduced in Section 2.4.

The actual Task Planner module is used to plan robot trajectory when user requires and action involving robot movement. The resulting plan is then passed down to the Execution Module. Upgrading the system to ROS is moving this planner to the robot itself, so each robot will be in charge of planning their own movements when some action is required from them. In this iteration the Task Planner is intended to be the component that should detect when the user requires assistance from the system, in real time as well as planned based on the current long term goal of the user. For InHANDS these long term goals are recipes that a user is willing to prepare.

The Task Planner is so far the less developed module and the one requiring more effort to put its functionality to the expected level in the system. Given the characteristics of the current scenario and the specifications of the system and it is where this thesis will focus its effort, offering an agent-oriented design to address this module.

**Execution Module**

The goal of the Execution Module was to control the execution of the robot and other actuator in the system. Once the robot control was updated to be ROS-compliant, this functionality was, as the functionality of the Task Planner moved down and encapsulated in the robot ROS ecosystem. While the module preserves the control over which and when robot capabilities are activated, it is no longer in charge of its low level control.

The Interaction Manager was the component in charge of maintaining a 3D representation of the scene, to ensure that the robot and human interaction was safe, modifying robot trajectory to avoid collision with the persons or any other object in the scene that moved after the plan was generated. This functionality is now integrated within the robot motion controller and no longer is needed. Therefore, right now, the main propose of this component is to interface with ROS robots control algorithm as well as some basic grasping policies to pick and place. Taking into account that grasping is a field of research for itself and the limited time the project has (two years) there is no aim to implement any sophisticated grasping technique.

In its current implementation, the Task Allocator consists on a monolithic piece of code that links the task issued by the Task Planner to the appropriate robot using the proper ROS interfaces. Then it uses the Interaction Manager to command the robot. Integrating new robots requiring to modify the application to take in consideration each robot, and implement, for each, its limits and the cost function that should be used to select one or another for a given task. Further more, to modify how this selection is done means to alter the code each time or to build an ad-hoc solution to configure the cost functions.
or the assignation algorithm. It also presents a considerable drawback in performance as it will not scale well as more robots are added, as cost functions could require planning and other intense computation for the robots. Although this module was out of the initial requirement of this thesis, it made sense to also rethink it. Task allocation generally has a close relationship with task planning and modelling it using the same approach will help so that it fits better with the new proposal for the agent-oriented Reasoning Module.

User Interface Module

The User Interface Module is currently suffering also important updates in this iteration. It originally possessed a pointer interface and a immersive graphical user interface. Now, it is being updated so it will include a few more interfaces such as speech recognition, sound output or motion(gestures) interface.

The main component, however, is an immersive graphical user interface used to notify the user about changes in the environment and allow user to command the system. This interface illuminates areas, shows interactive menus or notifications using a short-range projector. Figure 2.5 illustrates the main components and how they are interfaced. Figure ?? presents a snapshot of the interface in action.
2.4 Scope of this thesis within InHANDS

As it was introduced in the previous section, this Thesis will propose a new design for the Reasoning Module and the Execution Module. The following list enumerates the main functionalities to address.

- Integrate easily the new robots in the planning and execution pipeline.
- Given the actions the user is doing and the capabilities of the robots and other devices detect where the system could assist the user in his/her current activity.
- User needs to be able to ask for assistance if the required action is part of the system capabilities.
- Provide a fast Task allocation fast and select suitable robot for it.

![Scope of this thesis within InHANDS](image)

Figure 2.6: Scope of this thesis within InHANDS

Figure 2.6 shadows the scope covered by the work of this thesis. As the image suggest, this thesis will not deal with activity recognition, user interface. We will deal with the robots components as far as needed to interface with the robot using specified ROS APIs for each robot or component in the system.

Provided Interfaces

Table 2.1 specifies which inputs are given to the Reasoning Module and which is the expected output.
2.5. **DOMAIN SPECIFICATIONS**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current user activity</td>
<td>Ask User Activity</td>
</tr>
<tr>
<td>User commands</td>
<td>Ask For User Input</td>
</tr>
<tr>
<td>Current user action</td>
<td>Activate robot capability</td>
</tr>
<tr>
<td>Existing objects and its characteristics.</td>
<td></td>
</tr>
<tr>
<td>Result of robot capability execution.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Reasoning Module inputs and outputs

### 2.5 Domain specifications

#### User

The current system is targeting a concrete group of users. These users have the particularity that need to move around in a wheelchair. The users can be then divided in three major groups:

- No injuries or limitation in the upper trunk.
- Have limited mobility in one of the upper limbs.
- Have no mobility at all in one of the upper limbs.

Depending on the severity of their limitations and the side that is affected the working area of the user is limited. System actions like giving him an object need to happen within the user’s range.

#### System Actions

The robotic system is designed to perform the following actions:

- Bring and collect objects from or to specified positions.
- Give or pick objects from the user hand.
- Hold an object in a specific position and orientation.
- Pick an object and follow the user movement restricting the object orientation (i.e., cooperative transportation, bottle opening...).
- Aid performing some complex task (e.g., stir the content inside a recipient).
- Query the user for some input
- Notify relevant information to the user.
CHAPTER 2. INHANDS PROJECT

User actions that may be identified

The systems will provide as an input the action that is being detected each moment while the user is wandering in the kitchen. This will be a string identifying one of the following actions. The way the current activity is provided is out of scope of this thesis\(^2\). The Activity Recognizer still under development, and therefore, in this thesis its output are simulated. They will provide this as input to the system.

- Stir
- Pour
- Drink
- Smash
- Cut
- Blend
- Stripe

Monitored Activities

The system should be able to monitor the user when performing an activity and the progress he makes on it. To do so beside the result of robots actions we need to use as input the user actions that the Activity Recognizer successfully detects. For testing there were provided two different types of activities: Prepare a beverage or prepare a simple dish. To mention some examples:

- Prepare a soluble coffee
- Prepare a tea
- Prepare a french omelette
- Prepare a salad

2.6 Design Requirements

Their actual challenge is to detect the precise moments where assistance is need. We need to provide a solution that will integrate scene reasoning, monitoring, planing and task allocation. And do it all in real time, or as fast as possible. Table 2.2 presents the functional requirements of the Reasoning Module for InHANDS. In table 2.3 there is a list of desirable non functional requirements.

---

\(^2\)Activity Recognition is the subject of the PhD Thesis by Manuel Vinagre and Olga Mur, both members of Robotics Group at IBEC. At the time of finalisation of this thesis there were no reliable integration, and therefore the output of the Activity Recognizer will be simulated.
### 2.7 Proposed Approach

The system as presented forms a monolithic piece of software were robots and sensors have a high level of coupling inside the system. Because of that, modifications related to them or other parts of the systems (such as the allocation strategy) requires a considerable amount of work that has a very short life. That in turn is a potential source of issues that could break down the entire system. Furthermore it is expected that each component exhibits a smart behaviour, which is almost infeasible with a centralized commanding component.

#### Table 2.2: Functional Requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr.1</td>
<td>When user appears in the scene and is identified the system should start to monitor user</td>
</tr>
<tr>
<td>fr.2</td>
<td>System adapts his actions to a user profile</td>
</tr>
<tr>
<td>fr.3</td>
<td>When user goes out of the scene, if he gives some command, the systems will carry them on, store all the information gathered by the user and goes to hibernation waiting for another user appearance.</td>
</tr>
<tr>
<td>fr.4</td>
<td>Wherever a task is actionable, the system will try to assign the task to the executor that can carry it.</td>
</tr>
<tr>
<td>fr.5</td>
<td>Wherever a task could be performed by more than one agent, the one with lower cost should be selected</td>
</tr>
<tr>
<td>fr.6</td>
<td>CAPDI is able to carry objects</td>
</tr>
<tr>
<td>fr.7</td>
<td>When not asked to carry objects CAPDI could be asked to move its camera to a certain Y position</td>
</tr>
<tr>
<td>fr.8</td>
<td>Baxter is able to carry objects</td>
</tr>
<tr>
<td>fr.9</td>
<td>Baxter is able to carry objects with user</td>
</tr>
<tr>
<td>fr.10</td>
<td>Is able to hold objects at request of user</td>
</tr>
<tr>
<td>fr.11</td>
<td>Mico can move objects</td>
</tr>
<tr>
<td>fr.12</td>
<td>Mico can stir</td>
</tr>
<tr>
<td>fr.13</td>
<td>Robots that can move objects can collaborate to move an object from one point to another if there is no robot that can do it all alone but can do it concatenating their actions.</td>
</tr>
<tr>
<td>fr.14</td>
<td>User can issue direct commands to robots</td>
</tr>
<tr>
<td>fr.15</td>
<td>Actions in a plan can depend on other actions.</td>
</tr>
<tr>
<td>fr.16</td>
<td>Actions in a plan can be concurrent or completely exclusive.</td>
</tr>
<tr>
<td>fr.17</td>
<td>Actions in a plan can be optionals and not necessarily to be done to accomplish the goal. System should determine if an optional action should be activated.</td>
</tr>
<tr>
<td>fr.18</td>
<td>It should be able to manage different user profiles.</td>
</tr>
<tr>
<td>fr.19</td>
<td>System should generate necessary plans to archive the goal given the inputs and resources at its disposal each time.</td>
</tr>
</tbody>
</table>
CHAPTER 2. INHANDS PROJECT

Table 2.3: Non functional Requirements

<table>
<thead>
<tr>
<th>Id</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>nfr_1</td>
<td>The system should be scalable in the number of robots and inputs it can manage.</td>
</tr>
<tr>
<td>nfr_2</td>
<td>The system should be responsive, working in near real-time.</td>
</tr>
<tr>
<td>nfr_3</td>
<td>The robots and inputs are distributed in a LAN network and don’t need to reside in the same machine as the reasoner.</td>
</tr>
<tr>
<td>nfr_1</td>
<td>Should be easy to enhance the robots with new action and capabilities.</td>
</tr>
<tr>
<td>nfr_2</td>
<td>List of actions can vary with no need of major changes.</td>
</tr>
<tr>
<td>nfr_3</td>
<td>There should be reasonable easy to add new inputs and robots.</td>
</tr>
<tr>
<td>nfr_4</td>
<td>There should contemplate the possibility to be expanded with learning algorithms for user profiling, learn new activities or improve the given assistance to the user.</td>
</tr>
<tr>
<td>nfr_5</td>
<td>There should contemplate the possibility of being enhanced with assist-as-need control.</td>
</tr>
<tr>
<td>nfr_6</td>
<td>Should be possible to add communication mechanism to trigger events reasoning about user state.</td>
</tr>
<tr>
<td>nfr_8</td>
<td>Should be easy to expand the set of objects and concepts system can reason about it.</td>
</tr>
<tr>
<td>nfr_9</td>
<td>Keep ROS as a main development platform for robot capabilities and control.</td>
</tr>
</tbody>
</table>

in a system like that.

Given the domain specifications and the requirements presented for the Reasoning Module and the Execution Module it is possible to state an urgent need for reducing the coupling level of the application at the same time as the level of autonomy and intelligent for each component is increased.

This thesis proposes to address this scenario from a agent-oriented engineering point of view. Agent-oriented software presents as one of its most important advantages the capability to reduce coupling. It also presupposes that agents will be autonomous and, because of that, presents an easy paradigm to encapsulate functionalities in independent agents. We will use this case of study to explore multi-robot system implementation using BDI3 agents and provide some guidelines on how to do so. In Chapter 4, where we present a framework to do so in build smart, agent-oriented, multi-robot systems on top of ROS. Later, in Chapter 5, we present a entire design proposal for InHANDS Reasoning Module and Execution Module.

3Belief, Desires and Intentions, see 3.3
Chapter 3

State of the Art

This chapter briefly discusses literature from several fields that are relevant to this thesis. We start with an overview of what Assistive Technologies (AT) are, focusing later on examples where robotics and/or Multi-Agent Systems are involved in. In section 3.2 we give a brief overview on what Multi-Agent Systems (MAS) are. We focus specially in MAS development under Believe, Desire and Intentions (BDI) paradigm, which is discussed in Section 3.3. The last Section of this chapter introduces the basics of Robotics Operating System (ROS) and gives a brief insight into its main concepts.

3.1 Assistive Technologies

Assistive Technologies (AT) is usually used to refer to a heterogeneous range of devices and services used to palliate the problems that persons with disabilities have in their daily activities. Cook and Miller [CP07] provide the following definition of the term:

Any item, piece of equipment or product system whether acquired commercially off the shelf, modified, or customized that is used to increase, maintain or improve functional capabilities of individuals with disabilities.

Among other concepts in the first chapter of the book, a remarkable one is the characterization of Assistive Technologies as something different from rehabilitative or educational technologies. The main difference between the concepts is probably that Assistive Technologies are devoted to assist people in their daily routine. The other focuses on remediation or behavior reeducation to workaround or palliate the effects of disabilities. These technologies involve a wide range of devices, methodologies and practices that at the beginning
CHAPTER 3. STATE OF THE ART

were difficult to mix. Nowadays the frontier between rehabilitation, education and assistance is fading away, as high level technologies can address all of them in one single device.

It is important to note that when talking about persons with disabilities we also refer to elderly people. They are indeed potential users for Assistive Technologies. It is a fact that any person starts losing some of his capabilities at some point, and subsequently becomes a disabled person in some aspect or another. Population aging [Dep01] has been a matter of attention for a few years now. This is already a state matter in some countries. Japan is one of the first countries to address the issue and one of the leaders in advanced Assistive Technologies. They have produced systems like the exoskeleton HAL [TKHS10] or the caregiver robot Twendy-One [IS09] to mention a few remarkable examples.

More interesting examples of Assistive Technologies, more relevant to our field of study, are perhaps less related to a single physical device. There were projects that seek a more integral view of the situation, trying to enrich the user environment with things other than robotic devices.

A good example could be the RobotCare[CC11] project. It aims to integrate monitoring and assistive technologies in the home environment using a multi-agent approach. They have a sensorized home with one maid robot. The environment uses the Activities of Daily living (ADL) schedule set for a user to help him keep track of rehabilitation exercises, medicine doses and contact family or professionals in case of incidence.

On a higher level we can find Share-It [CAU08, CBM10]. The system is designed as a framework to support the development of devices capable of assisting user and his/her caregivers through mobility and assistive devices. They also use the ADL concept together with an ecosystem of agents working as an intelligent home environment to monitor the user activity and provide assistance. The results of the project were a series of prototypes like the i-Walker[ABC, CACL, BCA]. The i-Walker works with an ecosystem of agents that help to track the user behavior allowing to adapt and its performance learning from the user, avoiding dangerous actions and connecting with the caregivers in case it is necessary.

Using agents to operate and coordinate robots makes a lot of sense in our case of study. It shares similar aspects with RoboCare or Share-it. Probably the difference here is that the actual goal is to build a technological platform that embraces the scopes of these two projects at the same time. With that we mean that the idea is to have a standardized platform that helps for example, for example, to build an advanced assistive device using an agent-oriented framework such as Share-It to build a full ecosystem such as the one envisioned in RoboCare project.
3.2 Multi-Agent Systems

Until recently the way to program robot controllers at the highest level of abstraction was through Functional Programs. They could be depicted as a function \( f : I \rightarrow O \) from some domain \( I \) of possible inputs, that are mostly sensors, to some range \( O \) a possible outputs, referring mostly to robot actuators. Despite it is possible to find a wide range of well-known techniques to develop this kind of programs, some aspects of robotics programming, especially in the collaborative robotics area, start to make it challenging to define such programs.

After the functional layer of a robot that provides functional capabilities such as pick and place, moving or similar ones, the developer faces a far more complex paradigm. How to program a robot to be smart and find by himself how to combine its capabilities in order to solve the task? They have to maintain a long-term, ongoing interaction with the environment, reacting to its unexpected changes. When this is achieved you have a robot that will work by seeking the achievement of its goals, being this the definition often used for an agent. A relevant class of reactive systems that turns out to be well-suited for programming smart robot applications.

The concept of an agent is used in computer science to denote an entity that perceives and performs tasks in an environment in a more or less autonomous manner. Software agents can be opposed to agents that have a physical body like embodied agents (robots, smart factories), and biological agents (animals or humans) [FG97].

There has been much discussion about what exactly constitutes an agent [Cas97], and many definitions of an agent have been proposed [FG97]. In general, definitions that are acceptable for the majority of researchers are often considered too broad, but more specific definitions are usually only accepted by a small group of people. In this thesis, we will follow one of the most common definitions of agents [WJ95, Woo09]:

\[
\text{an agent is a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives.}
\]

Here are the properties Jennings and Woodridge argue that an agent should have:

**Autonomy**

Typical functional programs do not take the initiative in any sense, they just respond to the inputs. Roughly speaking our aim is to delegate goals to agents, which decide how to act best in order to achieve these goals.
Agents are autonomous as they encapsulate control, so that they cannot be controlled or invoked. An autonomous agent makes independent decisions about how to achieve delegated goals without being directly driven by others.

**Reactivity**

Robotic domains are characterized by highly dynamic conditions: situations change, information is incomplete, resources are scarce, the actions performed are not deterministic in their expected results. 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 behavior turns out to be tough.

**Proactiveness**

Agents are proactive by definition: 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).

**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, having more agents that fulfill a specific part of the overall computation could be useful to achieve a good level of work balancing.

Shoham introduced in [Sho93] the agent-based programming paradigm. Agent-based programming uses proactive agents as the main components of a program and, in contrast to object-oriented programming, where reactive objects form the building blocks of a computer program. There are a number of programming languages that support the development of agents and multi-agent systems [BDDS05]. Before we focus on the more concrete approach of BDI agents, this Section will finish with an overview of JADE, currently the most prominent and used agent middleware in the literature.

**Jade**

The JADE\textsuperscript{TM} agent platform [BPR99] is a middleware for developing and deploying agent-oriented software solutions. It complies with the IEEE FIPA standards. The agent system contains the main container (composed of the DF agent, the AMS agent, and the RMI registry). Additional agent containers may be launched on the same host, or on remote hosts (one container per one host), that connect themselves with the main container of the Agent Platform, resulting in a distributed system that seems a single Agent Platform from
3.3 BDI PROGRAMMING

the outside. Agents can migrate or clone themselves to other hosts. The strong point of Jade is the communication layer that allow to use multiple protocols and architectures being compliant with FIPA agent communication standards, making very easy to implement communication protocols based on them. JADE is distributed by TILab as a free and open source software under the terms of the LGPL (Lesser General Public License Version 2). It is a robust platform with a lot of documentation and used as a base for many different agent platforms like most of the ones mentioned in Section 2: BDI programming. This is the case of our platform of choice 2APL.

3.3 BDI programming

The BDI-based agent programming paradigm is based on Bratman's theory of human practical reasoning, in which human reasoning is described with the notions of belief, desire and intention [Bra87]. Rao and Georgeff were the first to formalize Bratman’s theory, and later, they developed a BDI-based software model [RGO95]. BDI agent frameworks try to simulate the way people think that they think. BDI programming has been and is being developed at universities and is currently mostly used in scientific settings. Still, there are some examples of practical applications of BDI-based programming. The use of BDI agents has been explored for programming robots before. For instance, list of papers. In this Section we will introduce a general BDI architecture. Subsequently, we will discuss in more detail 2APL, a BDI-based programming language as we will use these language to illustrate our approach and to implement agents for the practical scenario in this thesis. For a more extensive discussion of the BDI approach we refer to [Woo00], where Wooldridge presents a mainstream view on BDI agents.

BDI architecture

These is no single BDI model. The BDI approach is represented by a family of BDI architectures, each implementing its own interpretation of BDI theory. Examples of BDI-based programming languages are: PRS [GL87], Jadex [BLP03], Jack [BRHL99], Jason/AgentSpeak [BDDS05], ConGolog [DLL00], 3APL [HDdHM99] and its successor, 2APL [Das08], and GOAL [Hin09].

These languages have in common that an agent’s mental state is defined by its beliefs (representing the agent’s knowledge), goals (desires) and intentions (goals to which the agent commits itself). Usually, BDI agents also have a plan library containing a set of plans, where a plan is a recipe for achieving a goal given particular preconditions. The plan library may contain multiple plans for the achievement of one goal. An intention is the commitment of an agent to execute the sequence of steps making up the plan. A step can be an
executable action, or a sub-goal for which a new plan should be selected from the plan library.

![Figure 3.1: Overview of the BDI architecture.](image)

Figure 3.1 shows a general BDI architecture shared by most BDI-based programming languages (adopted from [Woo00]). The mental state of a BDI agent (the gray box in the figure) is constituted by its beliefs, goals, plans, and intentions in its belief base, goal base, plan library, and intention stack, respectively. A BDI agent can perceive and act in its environment (perception and action in the figure). The behavior of a BDI agent is generated by a deliberation process on its mental state, performed by the reasoner.

Deliberation cycles differ per agent architecture, but a typical BDI execution cycle contains the following steps: (i) perceive the world and update the agent's internal beliefs and goals accordingly, (ii) select applicable plans based on the current goals and beliefs, and add them to the intention stack, (iii) select an intention, and (iv) perform the intention if it is an atomic action, or select a new plan if it is a subgoal.

**The 2APL language**

2APL is a typical BDI-based agent programming language and allows for agent representations in terms of beliefs and goals. Moreover, 2APL is built in Java, which makes it suitable to extend the language with explanation facilities. In this Section, we provide a short overview of 2APL. For a more complete and detailed overview of 2APL we refer.

The mental state of a 2APL agent is defined by its beliefs, goals, plans, and reasoning rules. When the agent is executed, a deliberation process on this mental state determines the agent's actions. 2APL agents can interact with environments (e.g. the blockworld environment) by performing actions in the environment and receiving external events from the environment.

A 2APL agent can execute different types of actions: actions that add and remove beliefs to and from the belief base, actions that pass a message
3.3. BDI PROGRAMMING

to another agent, actions to interact with the environment, abstract actions encapsulating a plan by a single action, actions to test the belief and goal bases, and actions to add and remove goals to and from the goal base.

In 2APL, an agent’s beliefs are Prolog-like facts and rules, and the belief base of a 2APL agent is similar to a Prolog program. Thus, from the beliefs $x$ and $y : \neg x$, the belief $y$ can be derived. The goals of a 2APL agent are declarative, that is, they state what an agent wants to achieve, not how to achieve it. To reason with goals in the agent’s goal base, so called PG-rules (Planning Goal rules) are used, which are of the form $Goal \leftarrow \neg Belief | Plan$. Informally this means that if the agent believes $Belief$, then to achieve the $Goal$ it should execute $Plan$. An agent can adopt a new goal by executing the action $adopt(Subgoal)$, which means that the Subgoal is added to the agents goal base. A plan is a sequence of actions or subplans. to [Das08].

A 2APL agent can reason with plans through PCrules (Procedure Call rules), which are of the form $PlanA \leftarrow \neg Belief | PlanB$. If an agent has $PlanA$ in its plan base and $Belief$ in its belief base, it can execute $PlanA$ by executing $PlanB$. In the agents plan base, $PlanA$ is replaced by $PlanB$, which usually contains a set of actions and/or subplans. Figure 3.2 shows the deliberation cycle of a 2APL agent. The cycle starts with trying to apply all PG-rules. Subsequently, the first action of all plans are executed, and external events, internal events and messages are processed. Then reached goals are removed. Next, it is checked whether any rules were applied, plans were executed, or events or messages were processed in the current deliberation cycle. In case they did, a new cycle starts, otherwise, the process sleeps until

![Diagram of 2APL Deliberation Cycle](image)

Figure 3.2: 2APL deliberation cycle.
an external event or message arrives.

3.4 ROS: Robotic Framework and Middleware

Robots are complex systems able to perform tasks by processing data collected through their sensors and interacting with the environment through their effectors. Robot capabilities are limited to their programmer’s software. This normally attends to three areas. The first one related to low level control tasks such as kinematic modeling and motion control through the PID controller of an actuator. On the second level we can find sensing data processing to become aware of the world surrounding the robot. This level includes often mapping, navigation strategies or image processing tasks. The last layer defines how to use the sensor data and its actuators through a programmed behavior.

Developing a reliable robot with high level capabilities requires a team of specialists in many fields and a tremendous effort to bring together all the modules that form the robot. Because of that numerous robotic frameworks, middlewares and architectural proposals exist. Their main objective is to avoid reinventing the wheel and provide a proved base to build on, experiment and compare different approaches.

In this Section we will discuss briefly what a robotic framework, a robotic middleware and a robotic architecture are. As the final framework in use is already defined by the specifications given by InHANDS we will not discuss the available options in this area. Besides we will focus on the general characteristics of the Robotic Operating System (ROSD), giving the necessary insight to understand the work we have done on top of ROS.

Framework Middleware and Architecture in Robotics

The framework term usually refers to software that provides generic functionality at an abstract level and can be selectively changed by additional user code. A framework tries to be a reusable platform to develop new applications. Usually it provides support tools to program, debug and test such applications. Usually it also includes a set of conventions, methodologies and guidelines on how to develop applications for its domain of use. A robotic framework tries to simplify the development of robots software. Generally it follows a set of conventions and guidelines that ends defining the general architecture of the software.

A robotics middleware could be defined as a software glue that holds together all software components of the robot. It usually provides supplementary services to the ones the underlying operating system has, generally related to
process management and message passing mediation between software modules. A distinctive treat from a framework is that a middleware expects to be invisible to the developer, abstracting him from the need to know underlying interfaces between software and hardware.

In general frameworks and middlewares can not be distinguished in most of the cases. In fact a robotics framework is usually created to provide the middleware functionalities in addition to an API that will help the developer to integrate existing services and modules that proved to solve effectively general robotics problems like navigation.

A robotic architecture, on the other hand, is a more abstract description of how the modules of a robotic system are interconnected. It has to be defined in such a way that can be applied to a wide range of robotic systems. The selection of an architecture could have a huge impact on the type of robotic systems that can be developed. Generally the same solution can not be applied both to single robot systems and to multi-robot systems.

**Robotic Operating System**

ROS\textsuperscript{[QCG+09]} is a general purpose robotics framework and middleware that is widely spread and has numerous commercial products powered by it\textsuperscript{1}. It is a distributed system in the sense that the hardware it manages, the software nodes and applications can run in different machines and communicate with each other. From the network point of view, the process communication is done peer to peer, but at node level it requires a central instance, the roscore node, to be run on a machine known beforehand to all other machines hosting ROS nodes. This machine works as directory service and hosts the parameter repository.

ROS is by large the most used framework these days and has become the \textit{defacto} standard for the robotics community, especially for hobbyist and researchers. And it is starting to be integrated in the industry thanks to the \textit{ROS industrial consortium}\textsuperscript{2} that is backed up by big firms like ABB. ROS runs only over UNIX systems. Basically it is developed to run over Linux under the Ubuntu distribution. It is actively developed and has a wide community of contributors from universities, enterprises and individuals. And despite its notable limitations it can be easily interfaced to other middlewares to overcome them. For example when hard-realtime requirements are needed it is usually enhanced with OROCOS, when its network architecture is not totally suitable it can interface with YARP and be completely distributed. Although these interfaces are provided its implementation requires an effort from the developers of a specific application to make them usable. Its last

\textsuperscript{1}Robots powered by ROS: \url{http://wiki.ros.org/Robots}
\textsuperscript{2}ROS industrial consortium - \url{http://rosindustrial.org/current-members/}
release Indigo Igloo was released with 643 official packages. The packages’ functionality ranges from providing compatibility with most usual robotic and electronics hardware to high-level capabilities like Slam mapping or navigation through unknown environment. All these can be used, practically, out of the box if you use one of the dozens of robots that are compatible or directly powered by this middleware. This makes ROS the most suitable candidate to use as a base to develop multi-robots systems such as InHANDS. However it is lacking of any social aspect of robotics.

Architecture overview

The ROS architecture is based on a distributed graph of Nodes (processes) that communicate data using topics. Although the communication is P2P between nodes, there is required the execution of a Master Node. That node will act as address directory provider server and allows Nodes to establish communication channels to interchange data under specified topics. It will also serve as a central of data, call parameters, that Nodes use to modify their behavior, like a set of system parameters or environment variables.

ROS nodes could reside in the same machine or be distributed in different machines given the constrain that one of them acts as a Master and all the other uses that machine as directory to establish the communication network. The death of the Master node will not brake the network, but any other node that will try to register will fail and will not execute. The nodes that already run will keep communicating between them, but will fail to access parameters and this could affect their functionalities if they actively check for changes in dynamic parameters.

From the attempts to make it suitable for multi-robot applications there are a few initiatives like RoCon[SLK] as the most prominent one. Its approach consists on taking some capabilities that could be shared among robots like SLAM into the cloud so it is possible to share the computational cost and also actively joint the robots perceptions. While this is an interesting approach, it diminishes autonomy to the robots and requires developing orchestrating software.

One important aspect of ROS is its open philosophy. All in it is devoted to encourage package development as open source. They use not restrictive licenses for the main components allowing its use even for commercial applications. Developers can, if they want, restrict their package license to more restrictive license, for instance, not allowing commercial applications deriving from their work.
3.4. **ROS: ROBOTIC FRAMEWORK AND MIDDLEWARE**

**Node**

It is the basic functional unit of ROS. A naively definition, but sufficient, assimilates the concept of *node* to the *application* one. The difference is that here, besides the typical application inputs and outputs, a node generally makes extensive use of topics and messages in order to carry out its job. Examples of nodes are the ones that wrap drivers for specific hardware and exposes their data and functionalities through standard interfaces that then can be used by any other node implementing control functions based on such data.

**Topic**

A *Topic* can be seen as a message board. It is unidirectional in the sense that publishers just leave messages there without caring about who reads them. Its the way data is Shared in ROS. The data is encoded in special format types called *messages*. A topic then defines the type of data it will transfer and the name reference under it is transfer. As an example, a GPS driver node will be exporting the position through a topic of the type `sensor_msgs:point[]` under the name "position". Then, any other node that wants to access the GPS to get the current position of the robot will ask for the topic "position". This requires that the communications are defined at design time and it is complicated to establish dynamic communication in run time. This is one of the major problems to implement a MAS directly under ROS.

**Service**

Like topics, services are a communication mechanism. But this time they are bidirectional and synchronized. They are used to share functionalities. To put an example, a node driving a car would like to generate a route to go from point a to point b. Instead of encapsulating all these functionalities in the same node, we can implement the route generation in an independent node that will be providing the service "route_planning". This allows developers to later add other nodes that may use this functionality without the need of change a single line of previous code. Just register a client to the service, then request it using the proper message and wait until for the server node to send the response.

**Action**

An *action* is exactly the same as a *service* but it works asynchronously. That means that it is possible to make a request to an action server, but there is no need to wait until it sends us a response. So the process could be doing
other things while the action server computes the solution. As soon as it will be ready it will be send to the client. The action API\(^3\) provides also the possibility to cancel or to monitor the progress of server.

**Package**

Packages are the way functionalities are distributed in ROS. A package contains messages, services or actions definitions as well as nodes that make use of them to provide the functionalities it announces. It specifies also the software dependencies that the nodes have and automatizes the build process to compile the software from the source code.

\(^3\)Actionlib package -http://wiki.ros.org/actionlib
Chapter 4

The ROSAPL Framework

The lack of proper utilities and toolkits to develop multi-robot systems has been already mentioned in previous chapters. The IBEC’s robotics group is facing this very same problem. In Section 2.6, they need to integrate and coordinate a group of robots that will interact with humans in a simulated kitchen environment. This scenario has become a perfect case of study to develop our main ideas towards a framework for developing heterogeneous multi-robot systems. The result is what we call the Robot Operating System Agent Programming Language (ROSAPL).

ROSAPL is a set of conventions, guidelines and tools aimed to facilitate the development of heterogeneous multi-robot systems based on a BDI architecture. As the name may suggest the framework integrates on the one hand ROS, to provide easy robot development, and on the other hand 2APL platform, to add multi-agent support and reasoning based on BDI agents.

There were attempts to program robots using declarative languages in the past. [LRL+97] proposed a new logical programming language, GOLOG, to address the complexity of planning in a dynamic environment. [PHH99] proposed a Haskell-based language to address control loops and architectures. These approaches did not make impact in the robotics field, in our opinion, because they attempted to address too low level layers such as sensing or movement control. Industrial engineers tend to be reluctant to change things that work well, no matter how old they may seem. That was the reason why these approaches failed aiming a level of action planning, which was on the verge of being solved by what most of the community was proficient at: programming robots that sense, decide and move, through low level control algorithms inside microcontrollers in C.

We, therefore, took a different approach here. ROSAPL is not intending to replace the way robots are programmed at the lower levels. Moreover, it relies on the field’s state of the art, due to its interface to ROS. We think
that our proposal and approach addresses the levels of complexity that tradi-
tional approaches can not deal with properly. When a developer reaches a
certain limit of complexity in his system, he will search for a way to overcome
it. ROSAPL provides an upper layer relying on already proven algorithms
to control robot’s actions, and focuses on programming proactive and social
capabilities to define how a robot, or a group of robots, should act.

The framework tries to cope with the non-functional requirements of the
InHANDS project to allow future extensions of the project to easily reuse
components. It also provides guidelines and essential information to new pro-
grammers in the project. Therefore the current objective is not to have a
universal framework that could address most of the types of multi-robot sys-
tems, but rather one that is general enough to give flexibility to InHANDS and
will serve as a first step towards a future fully functional multi-robot system.

The current purpose is twofold. On the one hand, focus on design and
model high level control layers, more focused in decision making and planning
than fine control of robot capabilities. The other aim is to give a social layer
on top of that, allowing advanced human-robot and robot-robot interactions.
The main concern is to separate the what from the how. For example, the
idea is to find out if we need to go to the fridge and open the door rather than
how it is actually done. This lower lever is already managed properly by ROS
packages such as MoveIt\textsuperscript{1}.

In this Chapter we will discuss in detail the proposed architectures, models
and workflow. In Section 4.1 we briefly discusses what features the underlin-
ing technology provides towards multi-robot systems. Section 4.2 exposes and
defines the standalone concepts. In Section 4.3 the approach that has been
taken is described. Section 4.4 describes in detail the architecture and con-
structs of the framework. Finally, in Section 4.5 a review of the development
tools suitable to work with is provided.

4.1 Framework Bases

Building a robotic framework from scratch in few weeks is not feasible, spe-
cially if we aim at providing multi-robot functionalities. It has been proven to
be an enormous challenge that involves a lot of people and resources judging
from all the frameworks that we can find available. Moreover, the existing
ones are good enough, so it is unnecessary to start from zero. Therefore de-
signing and building a connection between existing frameworks seems a more
feasible and reasonable task to achieve our Master thesis goals. Combining
both gracefully we can try to almost cover all of them.

\textsuperscript{1}MoveIt! official page: http://moveit.ros.org
As it was exposed in Section 3.4, ROS is both an excellent framework and middleware. It has become a de facto standard for robotic platform development. Unfortunately, as any system, it has its limitations, the major one being probably related to its communication layer. The message infrastructure was not chosen and designed having in mind social robots. Because of that it is hard to make a group of robots to communicate among them properly and efficiently, using only ROS communication layer. The other one could be that if focuses too much on developing the robot capabilities and little attention is really put on how these capabilities are coordinated in order to have intelligent autonomous robots.

The problem at hand could be resumed as: Programming a Robot to act in such a manner that it is capable of detecting opportunities where its actions will help to accomplish some of its goals, while holding the capability to interact with other robots and humans.

The first concept that we can deduce from this statement is that a robot is an agent with goals. It also must be able to find a way to accomplish them. The means an agent has to fulfill a goal are the capabilities it possesses. A capability represents the sufficient knowledge to perform an action. The actions, when applied, will modify the state of the world in such a way that the goal is accomplished. It is a naive approach to think that there will be an action to fulfill each one of the goals an agent could have. It is more realistic to presume that the robot probably will need to execute a series of actions instead. That means that our robot should be able to plan which actions it needs to perform for modifying the world in order to accomplish its goals. For this, the robot, should also know the state of the world where it is. As it would be also naive to think that only our robot may modify the world, our robot should perceive events and determine whether these events should affect the current execution plan.

The previous paragraph partially introduces a possible BDI cycle as was presented in Section 3.3 and also the sense-update-act approach most of the robot are based on. Apart from the obvious reference to the concept of goal, the beliefs can be represented by the current state of the world, and the intentions can be seen as the commitment to execute an action on the basis of what it believes and in the behalf of fulfilling its goal. As we will see in Section 4.2, all the highlighted terms above are the key concepts that ROSAPL takes care of as a framework to incorporate the sense-reason-act schema to control the robot’s role. Because of that ROSAPL tries to be an integral approach to give personality and sociability to a robot in a methodological and practical way.

To provide a BDI-based reasoning cycle to the robot’s higher level of reasoning, we opted for integration with the 2APL platform. It is one the most powerful and flexible BDI implementations available thanks to its well-
balanced division between declarative and imperative programming. Among other reasons to choose 2APL, an important one for us is that it does not force the network and communication layers. Besides, we rely on Jade middleware to provide a complete messaging solution. If there is a need in the future, other messaging layers could be used, or even custom ones could be implemented. On the other hand, the platform is not production tested, meaning that there could be some performance issues. To minimize that, the execution of agents that could reside on the same robot can be distributed in different instances of the platform thanks to ROS.

Integrating 2APL into ROS does not only palliate the mentioned drawbacks, but also generates synergy between the two systems. As a result we have a P2P communication system between robots, assuming that Jade main container is publicly accessible. Being accessible allows any Jade system to communicate with our ROS, therefore opening other interaction possibilities to our robot in the future. We could have integrated Jade middleware directly with ROS to address the communication issues, but the way standard agents in Jade are programmed (as purely message-reactive ones) will not help to address social robotics issues. 2APL provides a shift in the programming paradigm available to our robot, giving more options to a robot developer.

The question is, perhaps, why there is any need of a framework and not a mere technological integration. The main reason comes out from the requirements of InHANDS project to have a modular and extensible solution that will be used for their assistive kitchen. To develop final MAS, as it will become evident in the next sections, 2APL is a raw platform. The particular case to find out how to implement properly a multi-robot system requires research, reasoning and conceptualization that is worth the effort to share with other people. This becomes a must if the expected users facing similar cases come from Mechanical or Electrical Engineering field with few or none formation in software engineering or advanced Artificial Intelligence. This makes it necessary to declare a common ground that helps using the MAS approach in Robotics by roboticists as well as computer scientists.

### 4.2 Framework Concepts and Models

Before going any deeper in the framework that is being proposed, we present in this section a definition of the main concepts that will be used to explain it. This is mainly to ground the terminology used to avoid semantic confusion.
4.2. FRAMEWORK CONCEPTS AND MODELS

Agent

The *agent* term generally refers to a software agent that will be normally embodied in a robotic platform. The general approach is to develop an agent for each robotic platform. Because of that in some cases the *robot* and *agent* terms are used interchangeably. In such a case robot is always used as reference to its controlling agent. In any case this is a norm, and in complex robots there may be different agents to control it. For example, one agent will be dealing with the platform and another will be in charge of interaction with the user. An agent in ROSAPL architecture consists of 3 layers (Figure 4.1).

![Figure 4.1: ROSAPL Agent programming layers](image)

The *Cognitive Layer* implemented using 2APL language, the *Operative Layer* based on the environment interface (Section 4.2), and finally the *Executive layer* involving hardware and low-to-middle layer control algorithms or advanced robot capabilities provided by ROS packages.\(^2\)

In general the agent will be implemented by a set of 2APL modules and a Java object that will define the interfaces the agent possesses to interact with ROS or other third-party libraries. The reader can refer the ROSAPL user guide in Appendix 6.4 for further reference.

Environment and State

The *environment* is the portion of the world that the agent can interact with. The environment is in fact the domain representation. It will be reflected as a set of beliefs in the agent so it would be capable of reasoning about them.

\(^2\)This is explained in more detail in Section 4.4, dedicated to the framework architecture.
CHAPTER 4. THE ROSAPL FRAMEWORK

To have a more manageable belief base we propose to follow the tendency in
the planning domain to represent the environment model as a list of facts.

\[
\text{[ ingredient(rice), tool(pot), at(pot, rice), at(table,pot) ]}
\]

It is possible that we represent the state of complex domains in different
sub-states to keep the reasoning process simple and efficient. We propose to
represent it as a functor that identifies uniquely a state or sub-state representa-
tion. This way it is possible to implement specific rules for each environment
or subset of it. The final form seen by our agents should be the following:

\[
\text{state( kitchen,}
\]
\[
\text{[ ingredient(rice), tool(pot), at(pot, rice), at(table,pot) ]}
\]

To facilitate the implementation of the domain ROSAPL provides two
constructs. The first extends the original 2APL environment to get access
to ROS features and directly call Capabilities as external action from 2APL
modules. The second one, \textit{State} is a useful class to implement the domain
representation. The \textit{State} basically a storage type where the programmer
puts all the domain items. If proper interfaces are used, the domain will be
able to propagate its changes to the belief base of the agents.

\textbf{Event, State Event and Message}

\textit{Events} are changes in the environment that the agent can perceive passively.
For instance, each time that the object detection pipeline detects a new object,
this will be perceived by the agent as an event that has modified the current
state of the world. This will trigger a change in the belief base, which could
then could break some preconditions to execute actions, and the agent may
be forced to re-plan.

We distinguish between standard \textit{events, messages} and \textit{state events}. The
last ones refer to things similar to the one exposed in the previous example.
Normal events are defined by the implementation. Messages and State Events
have a specific format when launched. Like in 2APL, events are caught using
Procedural Rules. Standard events are functors or facts sent by the environ-
ment that will require dedicated rules implemented ad-hoc for the system at
hand.
4.2. FRAMEWORK CONCEPTS AND MODELS

%message content is generally a prolog fact or functor
%but could be anything else.
message(sender,performative,language,ontology, content)

statechange(stateId,[new\_state\_list])

Goal

Goals represent the objective of the agent. They may take various forms: a belief we want to have, a relation that is wished to be achieved or a task that should be executed. 2APL has a good way of managing goals – they are stored in a dedicated knowledge base. We will use a set of conventions to declare specific types of goals and how to deal with them in order to take advantage of the modular capabilities of 2APL. The intention is to simplify 2APL programming and encourage module reutilization. We consider three different types of goals:

Belief Goal

A Belief Goal is normally represented as an atomic fact or functor.

goals:

    mapRoom.
    battery(100).
    mode(silent).

They will be normally used to achieve some internal state in order to modify the behavior of the agent. Following the example of the goal "battery(100)" will maybe hold other plans and force the agent to stop pursuing any other goal. This type of belief most of the times depends on the design problem at hand and usually are handled like a normal goal would be addressed in 2APL.

State Goal

Such goals express the will to achieve a new state of the environment. We model this type of goals as functors like state(stateId,[a]), where a is either a fact, a relation, or a list combining them. When an agent has one of these goals it will try to find a feasible plan to achieve it, adopting more goals if necessary.

Task Goal

This last type of goals are generally generated by the agent as a result of its deliberation in order to fulfill a State Goal or maybe a Belief Goal. By default they can be achieved in 2 different ways: by interaction with other agents and services or by executing an agent capability.
Action

The *Actions* are the known means by which the environment state could be altered. For us this is only another set of information that forms the knowledge of the agent and normally contains all the possible actions that could be done in a domain. There is no need for a robot or an agent to know how to perform them. It is enough to know the expected effects and the necessary conditions. This will allow our agents to reason about the environment and decide which actions are suitable to fulfill the goal. Later, given the capabilities an agent has, it will determine if it could be accomplished or not, and also to decide if it is worth searching for some other agent or service that could help.

There are several approaches to represent the actions in a given domain. STRIPS is the classical one, but it is quite restrictive in the type of actions that can be described. Action Description Language (ADL) addresses some of the limitations allowing to address more dynamical problems. But it also has several drawbacks, as ADL does not support the definition of optional conditions, execution preferences and constrains or plan quality evaluation. PDDL is probably the most complete one to describe actions and also the domain in general. PDDL is the approach we would ideally choose, allowing us to easily switch to more complex planning algorithm. The inconvenience of PDDL is its complex specification. Its integration would not be trivial and the effort to implement it would have consumed too much time. For this reason we decided to start using ADL.

The action:

```plaintext
Action ( 
    Fly (p: Airplane, from: Airport, to: Airport) 
    Precondition: At(p, from) 
    Effect: At(p, from) ^ At(p, to) 
)
```

could be encoded as a fact:

```plaintext
action( fly( Airplane,Origin,Destination ), 
    [at(Airplane,Destination)], 
    [not at(Airplane,Origin), at(Airplane,Destination)] 
)
```

Task

The *task* is the commitment of an agent to apply an action in order to fulfill a goal, or a set of goals. It may also specify some constrains such as time duration, repetitions, start time, end time or whatever the domain and reasoning
can handle. So we can define a Task $\phi$ as the action $\varphi$ to fulfill the goals $\psi$ under some constraints CONS.

$$\text{task}(\phi, \varphi, \psi, \text{CONS})$$

A task can be shared or delegated to other agents or services if the proper protocols are known. Generally the task is contained in an Action Plan. When the plan is executed the task goals become Task Goals.

**Action Plan**

We use the term *Action Plan* to refer to plans computed by a planner. This is to not confuse the 2APL notion of plan, encoded inside the rules and that conforms the agents Plan Base. \(^3\)

There are different approaches to represent a plan in the literature. A plan consists of an ordered set of steps, where each step is a unique operator instance. Plans can be totally ordered, in which case every step is ordered with respect to every other step, or partially ordered, in which case steps can be unordered with respect to each other. More sophisticated options present a similar representation to partially ordered plans using trees. Generally more expressive allowing to represent complex task relations specially designed for scenarios where we would like to enforce collaboration and cooperation.

Wurdel, Sinning and Frobig present a task specification called CTML[WSF08] that could represent a plan as a tree of tasks, where the leafs are atomic actions that can not be decomposed and intermediate nodes and root nodes represents compound tasks defining relations among its children. Doherty, Heintz and Landn present a similar representation called STS[DHL11], but as a recursive tree of task, where again leaf nodes are atomic actions, and root and intermediate nodes are defining how their children are executed: either sequentially, looping over them, in parallel or only one of them. We propose a mixed approach to represent plans that seems simple and at the same time expressive. We represent a plan as a directed graph where we have different types nodes that contains a task, as we defined it, and the edges represents the relations between tasks. The result is a graph (Figure 4.2 that resembles an inverted TST or CTML representations but avoiding intermediate nodes and simplifying the navigation, while preserving the same expressiveness, much richer than the traditional partially ordered representation.

Table 4.1 shows the possible relations we take into account between tasks. Table 4.2 exposes the types of nodes that we consider. It can be observed that

\(^3\)From now on we will use the term Plan to refer to an Action Plan, unless we explicitly describe other kind of plans.
CHAPTER 4. THE ROSAPL FRAMEWORK

Figure 4.2: Example of plan in the proposed representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Belief</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order Independence</td>
<td>If no relation exists between two tasks, these are order independent.</td>
<td></td>
</tr>
<tr>
<td>Choice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enabling &gt;&gt;</td>
<td>enabling(t₁, t₂)</td>
<td>Task 1 should be executed before Task 2.</td>
</tr>
</tbody>
</table>

Table 4.1: Types of task relations

the types of nodes represent unary operators on a task while the relations are binary relations that, excluding Choice, are transitive.

This type of plans could be generated by an agent using a planner, or be previously generated from other sources such as human instructions, or learned by demonstration.

Plans, as well as tasks, can have some constrains that may modify its computations or the selection of the execution strategy.

In ROSAPL an Action Plan π is a belief encoding a set of tasks φ₁...φₙ that have a set of relations ρ defining the graph, and modified by a set of constrains CONS.
4.2. FRAMEWORK CONCEPTS AND MODELS

<table>
<thead>
<tr>
<th>Type</th>
<th>Belief</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Necessary</td>
<td></td>
<td>If no type is assigned to a node, by default it is necessary.</td>
</tr>
<tr>
<td>Optional</td>
<td>optional($T_x$)</td>
<td>The task is optional and not necessary to accomplish the goal. All tasks that only depend on this task, are also optional.</td>
</tr>
</tbody>
</table>

Table 4.2: Node Types

\[ \text{plan}(\pi, [\phi_1...\phi_n], [\rho_1...\rho_m], \text{CONS}) \]

When an agent believes it has a plan, and assuming there is no other belief that restrains from it, it will attempt to execute the plan adopting each task as a goal. If it has the need or sees the opportunity it may delegate a task.

**Capability**

Until now neither the concept of action nor task have any effect on the world where our agents will live. They are reasoning concepts. A capability represents the know-how necessary to execute an action, and therefore, to be able to fulfill a task and achieve the goals by its own means. The scope of a capability is not limited only to actions. In ROSAPL perception and complex external actions such as elaborating plans using an external tool or service will be modeled as capabilities.

This means that, by giving to an agent a capability, we give it the belief it could perform such actions, estimate its viability, utility and cost given the current state of the world or whatever we need to evaluate whether to use it or not. Concepts used by most of the MAS design methodologies, like perception and actions, are modeled in our framework as Capabilities.

Thanks to the division in declarative and imperative parts that 2APL provides it is possible to contemplate the existence of relations between capabilities. Ines Nuñes[14] formalizes three relations: Association, Composition, and Inheritance. Figure 4.3 shows an example of such relations. The association means that a capability knows the goals of other capabilities and can issue them in order to accomplish his goal. The composition means that a capability is built upon other capabilities and it can access the goals and also the internals of each capability. Inheritance means that the capability is an extension of a set of capabilities, it has the same public interfaces (same goals as the parents), and also its own. Figure x illustrates that.
Figure 4.3: Example of capabilities implemented using relations, as shown in [Nun14]

At implementation level all these relations can be modeled in ROSAPL without introducing much complexity in the declarative layer of an agent. As we already explained, both perceptions and actions that an agent should posses are implemented also as capabilities. Because we rely on the ROS platform, perceptions will most of the time rely on topics which our agent is subscribed to. To implement actions extensive use of service and action packages will be necessary. Our framework presents constructs that help to wrap all this and concentrate on the logic related to manage a capability rather than how to interface it with ROS.

Capabilities are not limited to interface with ROS, they can be used to interface with any other third party functionality such as ontology or database querying. They can even implement directly the action. Therefore they are probably where most of the environment modeling and interfacing effort will be put. But also, if designed well, they will be the most reusable part.
Service

*Services* are standalone functionalities not associated with any agent in particular. It is possible that, for some reason, only a few have access to certain services. Services are generally provided by third party libraries, e.g., access a database or send and e-mail to say some. When a service is invoked it always tries to perform the functionality it provides, assuming proper parameters within the call are properly provided by agent. Capabilities in the other hand are only possible to activate from inside the owner agent.

## 4.3 Overview of the ROSAPL workflow

Integrating ROS, Jade and 2APL solves partially the most basic needs of a social robotic system. There are, perhaps, some elements that could have been made easier for developers to deal with. An anonymous multi-robot developers will have to decide how to deal with the following aspects:

1. World (domain) representation.
2. How to represent the possible actions in order to plan and how are they applied.
3. Planning toward an objective (goal) given a set of possible actions.
4. Decide which actions of the plan should be executed.
5. Delegate tasks to other agents, or accept tasks from them.

In this Section we present our approach to put some bases to solve these issues. In later sections we will discuss in more detail about each aspect and how to address specific concepts and tasks defining the whole system architecture. Figure 4.4 presents our proposed workflow for a general robot platform.

The idea is not too different from how, for example, motion capabilities are implemented. The difference is that we address when (what) to use this capability. At the ROSAPL reasoning level we do not care at all how is the action executed, but we care about the result, specially the reason because it failed, in order to adapt and recover from the error.

Our model then is a mix between Goal and Event driven. While goals is the main force directing the system (pro-activeness), the events are capable of changing them as major force (reactiveness). Therefore, the way a robot acts is determined by a set goals that the robot may face and a set of events that may occurrence at any given times. That is reflected by the inputs in the
Figure 4.4: ROSAPL approach workflow
workflow diagram goal, unexpected event and delegate task result. In general, messages could also be considered as events. While it is not reflected in the diagram, it is expected that events and messages might modify the belief base or even the goal base.

The particularity of our approach is that we take advantage of 2APL design and allow for multiple instances of this cycle to be running in parallel for a single robot, one for each goal it has. This allows the robot to maintain different plans in parallel, one per main goal in the system. Because of that way goals are included inside the plan belief, to allow only one cycle per main goal.

When a goal is generated it is adopted. The agent then tries to generate a plan to fulfill each one of its active goals. If it ends up not being capable to compute a proper plan, the goal will be dropped. This is because BDI agents in 2APL do not pursue goals that they do not believe are possible to achieve. If, on the other hand, it is possible to achieve, it will generate a plan an a set of sub-goals associated with it.

The plan should be evaluated to detect where the agent is capable of acting and where it will need to find partners. If it is not possible to find the appropriate partners, the plan is unfeasible. If partners are found, tasks are delegated and properly scheduled. Then each agent can start to execute its own task. While nothing external happens, after finishing a task, agent evaluates the result and, if nothing strange happened, he proceeds with the next task. Meanwhile, two other things can happen. The first one are unexpected events, that should be evaluated as soon as they arrive in order to see if they affect the plan. The other is to receive the results of delegated tasks. Both would require a plan evaluation and eventually require a re-planning or, in the worst case, result in aborting the plan and dropping the goal.

Our aim for this framework is to present a general approach to implement this workflow, that can be subsequently adapted to the requirements and restrictions of a concrete problem. The result of this thesis provides constructs and functionalities to manage actions, tasks, capabilities, and extends capabilities of 2APL for planing. We implemented a simple Means-End Analysis planner to serve as an example of 2APL plan generation enhancement to show the potential of combining the features that 2APL, ROS and the underlying languages (C++, Java, Lisp, Prolog and Python), and providing ways to enhance in the future the plan generation capacity with more advanced planners when needed.
4.4 Architecture

We propose an architecture that is divided in two levels. The first level refers to the agent internal architecture as shown in Figure 4.1. The second level devoted to the whole multi-robot system and its Social Layer, discussed in Section 4.4, that could include humans.

Each level defines a set of layers and proposes interfaces to properly build the cycle presented in Section 4.3. While the methodological approach to design of such systems is top-down, we will start to explain it bottom-up. That helps us to expose the way an actual robotic solution can be extended with our approach.

Robot Agent Architecture

We will start by explaining how the robot is envisioned, and how it is expected to be build with ROSAPL. When the developer has the blueprints of the system, the aim units in it will be the robots. They will hold some role to fulfill in the system, therefore, each one of them will have its goals, its inputs and its actions defined. We will use that to build up the robot in 3 layers.

The first one is the robot hardware layer and its controllers. There is no need in extending ROS to support this layer. Furthermore, in our opinion, a framework for multi-robots should be agnostic to the underlying robotic wirings in order to be extensible. Because of that section focuses on the Imperative Layer and the Declarative Layer.

In the next Subsections we will provide an introductory view of each layer. The interface between these two layers are the Events, External Action and Capability Action.

Agent Imperative Layer

Once we determined the inputs and actions (perceptions and capabilities) that our agents would require, we could decide to use any platform in the market that is powered by ROS, or build our own one. This will conform the ROS layer of the final agent. This platform will provide a ROS API based on topics, services and actions, which will serve to command the robot, for instance, to pick an object, or move it to a new location.

The imperative part of the agent differs a little from what 2APL offers. To implement it 2APL relies on a Java class, that should be inherited from to build the representation of the environment and all external actions available to the agents. While we do not restrict that approach, we add some more definitions and interfaces to standardize a little the development of these parts.
4.4. **ARCHITECTURE**

Hence, ROSAPL offers default constructs to represent the different types of agents, their vision of the world and their capabilities to interact with it. The environment then is left for very general purposes. Examples could be to synchronize environments of agent platforms in the same ROSMASTER. All the hard work is now to split among interfaces and constructs (introduced back in Section 4.2).

To properly define the agent that will control the robot we will start by enabling its access to the ROS API. Our implemented *Capabilities* will be in charge of that. Besides ROS-based capabilities, we could implement some as interfaces to any third-party package that will enhance our robot. For instance, access some on-line ontologies, learning algorithms or databases, perhaps an Internet directory to get addresses where to find shops to go and buy necessary goods or to avoid traffic jams.

To represent the different agents that the system may have, each one of them with a different set of capabilities, we propose to configure an agent specification for each one of them through the *agent* construct (introduced in Section 4.2). This will act as interface between the declarative part and the imperative part, allowing the execution of a capability decided inside a rule.

The last interface, but not the least important, is the *State* (also presented in Section 4.2). A State will be used to represent the current state of the world as seen by the agent, or a group of agents. It allows to simplify the way changes in the world could be reflected in the belief base of the agent, and therefore, affects its plan execution or decisions. At the same time it allows sharing a set of beliefs among agents if they access the same State instance.

Another contribution of ROSAPL, partially thanks to the Capabilities, is the simplification of the environment development in 2APL because there is no need to have the Environment class as the single entry point. Because of that, capabilities are directly callable without the need of any additional code to the final environment implementation, something that would be necessary in the standard 2APL environment interface.

**Agent declarative layer**

In our framework this is the most important part, as it simplifies the way to program the robot to take a decision, plan and act. It represents its mental state and its vision of the world. With that and the expressiveness that declarative programming provides, it is possible to achieve a smart robot acting without too much overhead. This requires a complete shift in how a robot program is envisioned though. Therefore to develop this part it is recommended to leave all imperative habits aside. This layer, as shown in Figure 4.5, is based on the 2APL architecture, but instead of contemplating a unique set of rules in a module we contemplate the possibility of having more than one.
Each of them is in charge of specific operations related to the flow presented in Diagram 4.4, and use belief, goals and Procedural Rules to define public interfaces so that each main module or submodules could be interchanged without affecting their related or dependent agent mental capabilities. This is in fact a general idiom taken from Object Oriented methodologies.

It is important to note here that there exists a 2APL extension, presented by [Das08], that provides a similar mechanism. Unfortunately, its implementation does not correspond to the latest version of the language, and has substantial differences from the current platform version. The effort and time to integrate it would not benefit the scope of this thesis, and therefore it was left aside.

Following those design lines ROSAPL proposes a set of module interfaces in order to implement the proposed workflow. Those interfaces propose a representation for possible actions, manage states of the world, generated plans and social communications. To manage incoming and outgoing messages we will rely on the communication protocol interfaces, that each agent of final system could adapt to use the contents they need. This is the reason why the design is top-down. So once we have all the protocols that apply to a set of agents, we could design a proper module architecture to address the agent’s role. We propose 3 basic pipelines inside an Agent. One is in charge of
4.4. ARCHITECTURE

Figure 4.6: Proposed archetypical module hierarchy and relations for a robotic agent

generating and revising plans, another will cover the execution of the existing plans, and the last will be responsible for delegating portions of the plan to possible candidates. Besides these main modules we also propose an interface to manage Capabilities in a standardized manner. Figure ?? allows to get the full picture of this approach. It is not expected to be the unique structure to support a robot reasoning and acting cycle, but it fits with the objectives and basic requirements for our case of study and as a proof of concept.

Social Level

This level conforms the multi-robot system as a whole. It is the solution for a concrete scenario or set of scenarios. It defines the final set of agents that will be needed, which roles they will have in the system and which actions they should be able to perform. This level is where communication protocols, norms, control mechanisms and other social aspects of the robotic system are defined and implemented.

ROSAPL makes no restriction in the way developers may define the different robots in the system, their roles and capabilities. We do recommend to use an Agent-Oriented Software Engineering (AOSE) methodology in order to design a multi-robot system that abstracts away from low-level details and is extensible to new actors and functionalities. In this thesis we will show
examples in Prometheus. Yet any other would have served our purpose. It is
a matter of the designer preference, whatever the designer is more proficient
in or feels comfortable to work with. ROSAPL has little to contribute to make

![Diagram of multi-robot architecture](image)

Figure 4.7: Multi-robot architecture

the design and development of this part easier. There are a lot of normative
frameworks to regulate social relations between agents. ROSAPL puts the
bases to work with them in a generalized approach. The idea is to use them
together with the design methodology of choice to meet a solution and imple-
ment it in a right away. In Figure 4.7 we present an overview of the proposed
architecture. Notice that all this can be enhanced including the notion of
Norms and/or control mechanism. This will facilitate to model interactions
between agents in complex environment or to regulate open systems.

We intentionally abstracted from the communication layer and do not
think it should be something to standardize. Robots sometimes have propri-
ety protocols, or work over industrial networks. There is also no need of
messages being fully compliant to FIPA message specification since most of the times robotic messages tend to be constrained by the efficiency requirements (i.e. using the minimal signal to pass the message), and most of the multi-robot systems do not require the overload open agent communication introduces in its messages.

Because of that we thought the message format that 2APL uses is both, simple enough to be used over constrained networks and expressive enough to build upon it complex communication protocols and interactions when the need arises. It there is the need 2APL API allows the development of custom messaging layer through inheriting some provided interfaces and constructs.

In Diagram 4.7 we propose to run each agent in its own platform. 2APL loses efficiency when many agents reside on the same platform. Each robot also has its own roscore running so the communication between the robots is entirely done through the communication protocols and messaging system. Running under Jade middleware, these communications allow us to add and remove agents without the need to touch any other part in the system. This is something not possible to do if you instantiate a MAS system using 2APL, as it has no way to add agents to an already started container.

The main contribution of ROSAPL in this layer are utilities referred to standardize the way to launch and configure an agent container. The interested reader can find some examples in Appendix A.

Human-Robot Interaction

One of our main objectives is to not only design a multi-robot system, but also make it able to integrate humans. In the architecture (Figure ??) we already included an overview on how a human will interact with the system. In this section we present various options to integrate a human in the system. Depending on the level of inclusion the approach might differ.

Robots that interact with humans at interface level

In this scenario we might want our robot to interact with persons while it is wandering the world doing whatever its commitment is. In such scenario there is no need to incorporate the person as an agent in the system. As the interaction is done at local level the best option is to dote a robot with means to interact with a person. Solutions such as voice recognition, speech synthesis, face recognition and motion tracking can become a powerful human-robot interaction interface.

In such a case the interaction interface is implemented through robots perceptions, capabilities or third party solutions like, for example, text to
CHAPTER 4. THE ROSAPL FRAMEWORK

speech synthesizers. Diagram 4.8 shows a possible configuration. From the

Figure 4.8: Overview of possible human-robot interaction system as a robot interface

MAS perspective the person is a part of the environment that can be sensed and interacted with. The only aspect of the system where the person is taken into consideration is in the development of the User Interface at robot level, which is how normally it has been done.

Besides ROSAPL is intended to help building multi-robot systems it will apply equally to design single robots to define the way it will act depending on the external stimulus and observations. This also has the advantage that predisposes the robot to be integrated in the future inside a social robot system.

Robots and humans cooperate or collaborate

In this scenario the robot and the human are no longer communicating, they are interacting, even working together to achieve a set of goals in a given situation. In such case, relying only on an interaction interface is not enough. It is necessary to have a model of the person and monitor its activity. This way it will be possible to adapt the robot actions to cooperate with the human.

The situation requires that the person is integrated in the environment, being an active actor in the system. The communication between a human and robots is then done through an avatar agent that implements the proper user interface as a capability. This agent is in charge of maintaining a set of beliefs about the user and communicating with other agents through established communication protocols.

In this case, the person is a part of the MAS system design and it is taken in consideration in the whole design process, defining the roles it plays,
4.5. DEVELOPMENT CYCLE

which capabilities should have and in which communication protocols should interact.

This is the principal type of scenarios where Assistance Robotics resides. In general it is not only the moment human and person are together, but also when not, that the robot should have in consideration the person in order to act. Moreover, in the new generation of AT there will be the interaction of caregivers, health professionals and patients with the technological devices, wherever they will be robots or just software solutions. And this is the ground where MAS perspective can provide more advantages, and where ROSAPL could become a integral solution.

Figure 4.9 shows a possible system configuration for InHANDS. The kitchen has the capability to detect when a person has fallen. It might have an autonomous wheelchair or walker will move with the intention to help the person to stand up. Meanwhile the system does not receive any evidence that the person is moving, or on the contrary, he is screaming and asking for help. The system may contact the caregivers or/and directly the emergency center so an ambulance is dispatched as soon as possible.

Figure 4.9: Overview of a possible assistance system based on agents

4.5 Development Cycle

To summarize the design process here is a brief list of steps.

1. Use a MAS design methodology to transform the requirements into an
actual MAS specification.

2. From this MAS specification define the actual platforms you will need, and which capabilities should they implement. Define the ROS API for each one of them.

3. In parallel to robot design or programming, the work on the agent layers may start. As the capabilities and the ROS API are specified, it is time to define the domain definition and the environment implementation.

4. Once the domain is defined and the Capability API is also defined, it is possible to start to work on the declarative layer, implementing the required modules and communication protocols.

5. The final stage will be to generate the necessary configurations for each robot in order to properly launch the rosapl_runner and start working.

**Development tools**

To assist developers there has been no need to develop new tools. There are already existing tools that will be useful to develop using ROSAPL. To do so we divided this section in two. The first one devoted to development tools and the second to execution monitoring tools.

**Code development tools**

The languages that can be used are the major ones: C++, Python and Java. However, when developing the multi-agent side for now only Java is expected to be used. Developers can choose to develop in any IDE they prefer. But Eclipse covers all of them in one way or the other. Even 2APL offers an Eclipse plug-in for its syntax. Unfortunately, it seems to be out of sync with the last version of the language, so it does not work at the time of writing this document.

Eclipse also provides tools for debugging and testing the code. For Java it integrates JUnit test framework. For C++ ROS provides easy GoogleTest and BoostTest integrations and for Python there is the Nosetests suite.

ROS also provides its own tool, called RosTest\(^4\) to write and execute tests at node level and debug the interaction between nodes.

\(^4\)ROSTEST site:https://wiki.ros.org/rostest
4.5. DEVELOPMENT CYCLE

Monitoring tools

Having access to tools that help you to develop code faster is very handy, but there is also a need to monitor its execution, to be able to know the current state of the system. This is especially useful, if not required, in distributed and complex system such multi-agent ones. Currently, there is no other way to debug and check communication protocols, agent behaviors and other performance issues other than executing the application and peek at what is happening inside (except for logic checking tools that can be applied to 2APL programmed modules).

ROS has tools to monitor all its constructs and interfaces, and even provides a configurable GUI that allows developers to monitor and interact with all major parts of the ROS infrastructure such as topics, services and parameters. 2APL also provides its own monitoring GUI that logs the BDI cycle and the major aspects of the agents in that container. It allows to debug the multi-agent execution effectively. The problem is that it is not designed to work in a distributed environment where containers are located in different machines, and running a GUI on each of them may not be feasible. Fortunately it is possible to workaround this by using the Jade GUI as an external tool, allowing us to use all the tools this framework provides. Section ?? gives some indications on how to do it.

This set of tools requires a little more understanding of both base frameworks, so its recommended to look at its documentation first. With a little effort it is possible to set up a unified solution like using an integrated Eclipse project, if the developer prefers it. Therefore, there are different ways to combine the existing toolkits to cover the developer needs, making not a critical need to develop anything else.
Chapter 5

InHANDS powered by ROSAPL

The scope of this thesis within the InHANDS project was already presented in Chapter 2 Section 2.4 of Chapter 2. Figure 5.1 presents a more detailed view of the modules that have been developed and improved. The Reasoning module was lacking of a proper Assistance Planner and Plan Monitor. Therefore are new implementations. The Execution module will be distributed among robots so the Task Allocator module will be simplified and each robot will be computing the utility value for a Task only if he can handle it. In this way we decouple the system into pieces that can be run in parallel or distributed with not much difficulties as each User will have their own planing and task allocation pipeline.

In this chapter we present the proposed design solution to address the implementation of reasoning and task allocation and execution requirements using the ROSAPL framework introduced in Chapter 4.

The main task consists in addressing the reasoning module. It should be capable of generating an assistance plan for all the tasks that should be performed depending on the user profile and the capabilities of the robots in the system in order to help the user accomplish the recipe that was selected. The execution module will then take that plan and execute it managing different robots reporting back its success or failure. This feedback will be joined to the actions the user is carrying to monitor the progress of the recipe.

Before work on this thesis started, the approach was to build two processes, one that collects user data and create or regenerate a plan of an assistive task that will then be passed to the second process in charge of allocating a suitable robot under its control and initiate the execution of such task, possibly controlling more than one at the same time. This approach presents several drawbacks, though. First of all, adding new robots or capabilities will
require to implement a plug-in system, or something similar, to avoid rewriting or modifying several parts of the program to accommodate a new capability in a robot, a new robot or a new functionality in the system. Moreover, a monolithic approach will not scale well in a scenario where there are several users and/or many actuators to be controlled and interfaced. This will also probably require complex configurations to work across networks.

To address those issues we propose a MAS approach. We already stated in Chapter 3 its inherent modular and distributed approach benefits and how it predisposes each component to incorporate smart and proactive behaviours. Assuming we respect the communication protocols and internal agent interfaces that the system defines; adding new agents, functionalities or interaction protocols should only require few changes in existing modules. We will apply the ROSAPL framework described in Chapter 4 to implement the scenario.

Our proposed design is developed using the Prometheus methodology[PW05]. The initial Section of this Chapter introduces the main aspects of the methodology and its application. Latter we will describe the main aspects of the proposed design, leaving as reference for further details the complete design in the Appendix 6.4. In the design we will not go much into implementation details, the interested reader can find in following Section 5.5 and previous Chapter 4 how a multi-robot system implementation could be addressed.

5.1 Prometheus Overview

The Prometheus[PW05] methodology defines a detailed process for specifying, designing and implementing agent-oriented software systems. In addition to a detailed design processes (and many practical tips derived from previous design experiences), it defines different design artifacts that are produced along the way. Some of these artifacts are kept as documentation, and some are only
5.1. PROMETHEUS OVERVIEW

Temporal to use as stepping platform. Some of the artefacts are graphical while others are structured text (i.e., forms). Prometheus artefacts relate back to the agent concepts that were introduced in the previous chapter and it is specially useful for designing BDI systems. For example, actions and percepts are captured in the system specification phase; the detailed design phase results in what later will be 2APL plans, events and beliefs.

A remarkable aspect about Prometheus is its unordered approach and recursive refinement that makes it more agile and flexible. While the methodology suggest an order in the steps it is not enforced, and the designer jump one to another while refining the artifacts on them. The main idea is to improve the initial specification in a cyclic way until a coherent and adequate system is defined with full detailed descriptions that can be then implemented in the platform of choice.

The Prometheus methodology consists of three phases, depicted in Figure 5.2, where rounded boxes represent elements of the final design and normal boxes specify intermediate information.
CHAPTER 5. INHANDS POWERED BY ROSAPL

System Specification

The system specification phase focuses on identifying the goals and basic functionalities of the system, along with inputs (percepts) and outputs (actions). The results of this phase are:

- Identifying the system goals.
- Developing use case scenarios illustrating the system’s operation.
- Identifying the basic functionalities of the system.
- Specifying the interface between the system and its environment in terms of actions and percepts.

Architectural Design

The architectural design phase uses the outputs from the previous phase to determine which agent types the system will contain and how they will interact.

- Deciding what agent types will be implemented and developing the agent descriptors.
- Capturing the system’s overall (static) structure using the system overview diagram.
- Describing the dynamic behaviour of the system using interaction diagrams and interaction protocols.

Detailed Design

The detailed design phase looks at the internals of each agent and how it will accomplish its tasks within the overall system. This will serve as a starting point to implement the final system on the agent development framework of choice (in our case 2APL and the basic ROSAPL presented in Chapter 4). This phase defines:

- The refinement of agents in terms of capabilities, giving the agent overview diagram and capability descriptors; and
- The development of process specifications.
- Design of the plans within a capability and the events generated and handled by these plans, as captured in the capability overview diagrams.
5.2 INHANDS SYSTEM SPECIFICATION

- Specification of the algorithm within each plan, as well as associated data (or beliefs) and detailed specification of events. These are captured in plan, data and event descriptors.

5.2 InHANDS System Specification

We started the design process from a simple text description given by the members of InHANDS project:

We would like to develop a system to assist handicapped people in the kitchen. The system offers assistance through robots and an immersive user interface. The system should be capable of detecting where user need assistance in real time and execute secure and reliable assistance actions to help the user prepare his recipes. The system should also allow the user to ask for direct actions, such as pick and object, throw an empty bottle to the trash or stir the contents of a pot.

We will abstract from the current implementation of actions, as we assume they will be implemented as ROS nodes that will provide such capability using a ROS interface. This will simplify a lot system actions to a simple abstract one, after defining a proper interface for capabilities. We also assume that the actions that a user is performing in a given scene are continuously monitored and supervised by an external process that will notify the system about the performed action of the user as soon as it is identified.

After some loops in the first stages of Prometheus methodology a first set of Objectives and scenarios come out of the design process. A detailed description of the final design can be found in Appendix 6.4.

The goals, represented with the Goal Overview Diagram in Figure 5.3 illustrate the main goals of the system. We built the relations between goals mainly by asking how and why. Basically the how’s will determine subgoals and why’s will determine parent goals of the queried node.

There were defined 6 scenarios for the system:

- User Enters Scene
- Select Recipe
- User is Cooking
- Pending assistance
- User Task Request
- User Leaves Scene

From the set of goals and scenario there were determine a set of system functionalities that were then formalized into roles or functionalities inside the system. These are:
Taking into account the specifications given by InHANDS and the work made so far in the system specification a list of actions and perceptions was determined. When defining in other phases more specific models, agent capabilities and protocols some perceptions and actions where expanded to be more specific. The final inputs and outputs list are presented below:

**Percepts:**
- User Login
- Selected Recipe
- Recipes Query
- User Action
- Capability Success
- Capability Failure
- Timeout

**Actions:**
- Get Recipe
- Search Recipes
- Update User Profile
- Get User Profile
- Execute Task

Table 5.1: InHANDS Percepts and Actions
5.3. **ARCHITECTURAL DESIGN**

As final part of this phase, and again combining what was specified by the InHANDS team, the data sources that will operate the system were determined. For simplicity, Table 5.2 lists some data sources that are generated and managed outside the system. The rest of data, illustrated in figures all along this chapter is omitted here and can be found in the Appendix 6.4, where a more extensive listing and description is made.

<table>
<thead>
<tr>
<th>External Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserDB</td>
<td>User registered by the system and its medical profile</td>
</tr>
<tr>
<td>RecipesDB</td>
<td>Available recipes in the system</td>
</tr>
<tr>
<td>ActionsDB</td>
<td>Know actions that can be applied in the scene, including the ones recognized by the activity monitoring module seen in Figure 2.3</td>
</tr>
<tr>
<td>Capabilities</td>
<td>Set of information that make an agent to acknowledge which Capabilities it have, it depend on each robot.</td>
</tr>
</tbody>
</table>

Table 5.2: InHANDS Data Sources

5.3 **Architectural Design**

Figure 5.4: InHANDS System Overview Diagram

All the details formalized in the System Specification phase were materialized into final agent types and interactions between them. They are presented
in the following Subsections. We will reserve the relevant detailed explanations for the next Section, and in this one focus on general aspects of the agent types and interactions. An overview of them is described by diagram overview in Figure 5.4. This diagram shows the final agent types with their respective percepts, actions and relevant data. The double arrow labels represent the interaction protocols between the agents.

![Data Coupling Diagram](image)

**Figure 5.5: Data Coupling Diagram**

In order to better explain how we grouped functionalities into agent types we created a data coupling diagram(shown in Figure 5.5). This diagram helped to identify related functionalities that share data. To define the agent types we took into account these data relations and how similar are the scope of functionalities.

*Assistive Planner*, *Task Allocation* and *User Monitor* are functionalities very related one to each other and works User level. While Environment Manager has also access to *UserDB*, it make sense to have only one instance to manage the entire scene and identifying Users. We expect to have different robots that will provide their capabilities to the system, as there is no shared data between the capability provider and the other defined entities, it is logical that each robot have its own agent.

**Agent Types**

The role grouping diagram (shown in Figure 5.6) presents the final agent types and their corresponding functionalities or roles. As the system should
personalize the actions depending on the current user we opted to encapsulate
the user monitor together with all planing related functionalities. What it was
left is, in one hand the capability provider and in the other the Environment
manager. These 3 final agent types are: Avatar, Scene and Robot. Each
Avatar representing the user interface and model, the Scene Manager serves as
directory for system components and, Robot, the robots capable of providing
assistance to the user.

Avatar agent

This agent type is responsible for maintaining a model of the user, with his
profile and expected intentions such as which recipe is cooking. As the current
system will only provide one interface and will only handle one user, we mod-
elled the avatar as the agent directly in charge of the user interaction. This
also provides the advantage that if the system could present some interfaces
restricted to the user in a phone or tablet they can be easily integrated in the
agent.

Figure 5.7 shows the diagram for this agent. As is shown, besides the user
representation and interface, this agent also generates assistance plans. These
plans take into account the medical profile of the user and adapt the capabil-
ities of the current existing robots to provide the right amount of assistance.

The Avatar is also in charge of delegating those tasks and process the
outcome by updating the user plan accordingly. The update of the user plan
also happens wherever an action fitting the plan is performed by the user.
This could have as consequence the invalidation of some assistance task, or
the generation of new ones.
CHAPTER 5. INHANDS POWERED BY ROSAPL

Figure 5.7: Avatar Design Diagram

Robot

This agent type wraps an agent platform and interfaces with its capabilities to command where to execute them. The agent implements the policies for starting the execution of an action and the conditions under which a new action could pre-empt the current ones. As all the security is implemented inside capabilities at the ROS layer, the robot agent should manage to receive the result for the active capability and manage errors properly. For instance, if it should move to a concrete position but the space is not free, the motion plan will fail; this is captured as a percept (service or action client error response) and depending on the environment and pending task it will decide to retry,

Figure 5.8: Robot Design Diagram
or just inform that it is impossible.

**Scene Agent**

This agent type is in charge of user detection and serves as a Directory Manager for Capabilities. Wherever an user appears will ask for login. Once user login will be assigned to an Avatar agent. If there are no available avatar agents, it will create a new one.

When an Avatar is searching for a provider, the source to acquire this information is the Scene Agent. That will response with a list of candidates for a requested service.

**Interaction Protocols**

In order to allow coordination between different processes there were established 5 interaction protocols that are briefly described below. The reader can refer to Appendix 6.4 for detailed descriptions of the protocols and messages involved in them.

The main interactions between agents in the system are:

**Assign avatar** (Request)

Make an Avatar to represent a specific user that is detected in the scene and logged in.

**Release avatar** (Request)

Make an Avatar to release the user and wait for a new assignation.
Register Capability Provider (Request)
Robots will register capabilities at start up to make them available for interested avatars.

Request Capability Provider (Request)
Avatars will query the Scene to obtain a list of providers for a specific task.

Contract Capability Executor (Contract Net Protocol)
This is the most complex interactions and implements a FIPA-compliant Contract Net Protocol. First the Avatar will make a call for proposals to all know providers. The interested providers will submit a Proposal before a deadline. The rest will respond not interested or will time out. Once all candidates submit a proposal, or the deadline is surpasses, wherever comes first, the avatar will select the most suitable provider for the task to assign. Then it will notify wit a reject all the other proposals and accept execution from the winner robot. This protocol is used wherever the task is automatically generated or comes from the user interface.

5.4 Detailed Design
This section presents some of the relevant aspects of each agent and their capabilities. Most of these capabilities can later be mapped to a 2APL module where plans will be substituted by Plan Goal and Procedure rules while alternative path will correspond to Plan Repair rules. We also suggest to implement each communication protocol in its own.

The main capabilities of the system are the plan generation, plan monitoring, task allocation and task execution. Because of that we get into them in more detail as examples. The first one takes into account the state of the world and generates (when required) a plan to achieve the main goal, i.e., to cook something interacting with the user. The second takes this plan and should decide which of the actions can be performed by the system, an which should be left to the user, thus generating a set of pending tasks for the system. The goal of task allocation functionality to allocate all tasks to the most suitable executors at the right time. Finally the task executor should need to control the execution of each capability the agent posses. Each one differs in their main objective and serves as a base stone for a more advanced solution to that particular objective that can be implemented later.
5.4. Detailed Design

Plan Generation

Figure 5.10 shows the suggested capability definition for the current requirements. The main idea of this module is the following. Once the agent knows the recipe the user wills to prepare, it retrieve its plan from the database, then determine which optional steps may the user prefer to execute (personalize the recipe). Later it will determine which objects in the workspace should be retired and which ones should be added, inserting this extra actions at the begin of the plan. It should decide when an object will be not longer used to mark it as disposable if the user puts it in the limit of its workspace. The workspace preparation part uses a MEA backtrack planner to determine which objects are needed in the scene and which of the available actions to the system will put them in the appropriate place.

![Plan Generation Capability](image)

Plan Monitoring

Figure 5.11 shows the capability in charge of keeping track of the plan execution, detecting possible changes on it and finally deciding which task require assistance from the system and in which grade. This is done by determining which part of the plan could be carried out by the system capabilities.

Using the user profile these actions will be adapted to the user limitations and added as pending task. The plan progresses as the user realizes its actions. This is detected by an external tool that broadcasts the detected actions. Whenever a new event is raised the monitoring plan checks to determine if it corresponds to a plan task. If it does not correspond, it might ignore it. In the case that it will be a valid action for the plan it will be consumed, a newer task will be available and the system will be able to delegate it.
If the input is a task result the process is the same, with the difference that, when something failed, it takes the time to analyse the failure reasons, check the plan consistency and require replanning (if needed). A third event could happen, a spontaneous change in the environment, not predicted by any action. This will force to check the plan and maybe replan from the new state to the desired one, resuming the preparation of the meal or beverage.

**Task Delegation**

This module is an example of implementation of a negotiation protocol. Whenever there is a task to allocate it will try to find a suitable executer for each assistive task. Once it is selected, it will delegate the task with the proper constrains to ensure proper time execution. Figure ?? shows the capability logic to manage this protocol.

Where the mechanics of each step could be varied, and each user could even have different ways of allocating tasks specified as a parameter at runtime. Or could be expanded to implement more than one method to allocate tasks depending on environment and system conditions.

**Task Execution**

This is an example of interface to abstract the exact details that implement each capability, as they are intentionally left out of the Agent layer. The task execution capability will keep track of active capabilities, when to activate them and retrieve the result from, in our case, the ROS layer to evaluate if
the actions succeed or not. Once finished the execution, it will notify the contractor about the result. If a new request for the capability arrives, it should evaluate if aborts the one in execution and replaces it depending on the capability interfaces that ROS provides.

Each robot capability will be implemented in a different module. And later all this modules will be inherited by the Capability Executor module in order to serve as entry point to process delegated task.
5.5 ROSAPL InHANDS Implementation

This Section introduces implementation examples of various constructs proposed in this design using ROSAPL. We present here code snippets that should give to a developer a reasonable picture of the implementation details for the proposed design. We also describe examples and suggestions for designing the following constructs: Plans, Percepts, Actions, Message, Capabilities, Plans, Protocols and Agents. The order is not trivial as we will present the constructs starting with the more simple and reusable ones and finishing with the more complex ones such as a Protocol or a Role. These last two are normally aggregating or managing the other simpler components. Agent is a special case where normally mostly of its implementation is already done while all the other components were built following a designed interface that finally work together as an agent.

Data

To start the implementation before we evaluated the Data, wherever is external or internal to detect if there where some data missing that need to be simulated. We already stated that there is some external data that the system do not have control over it. There also some critical data in order to test the algorithm that should be simulated to be able to work not necessary in InHANDS lab.

The data that should be simulated is:

- User enters in scene
- User leaves the scene
- User action

These data is generated through text files and fed tot the system using ROS timed publishers that reads the files with some trusted reader such 2APL.

Plans

This is probably the most specific component in Prometheus, and the one that maps directly to the homonym 2APL plan. Prometheus refers to plans as the predefined way to deal with incoming messages or events to a capability. The way to define the agent behaviour, when they arrive, depend on how many plans we defined to deal with them. Letting the plans to be simple and straightforward. In 2APL this is what plans are. The specific content of any rule that will be added to the Plan Base and executed one step at a time.
So any plan that we define using Prometheus will correspond to the body of one or more 2APL rules that will share Belief and the trigger Goals. Percepts and Messages present examples of rules and possible plans contained on them. Plans will generally involve modifying the agents beliefs, goals or activating abstract or external actions.

**Percepts**

As we use 2APL as base platform, the Percepts map directly to the Events construct for that platform. This type of constructs needs to be implemented in both 2APL layers, in a module and in an accessible environment for the running agent. For instance, whenever an agent appears in the scene the *Scene Manager* will be notified using an Event. Generally this involves to elaborate a Procedural Rule that manages the specific event. With this implementation wherever a *newPerson* event is thrown, if person $P$ is not in Scene Manager internal data loggedUsers the agent will adopt the goal to identify that person. As there is no other rule, if the user already is logged in the rule not fire and the event is drop.

Snippet 1 presents how the Scene Manager will acknowledge that a new user entered in the scene and adopts the *Identify User* goal.

**Action**

Actions are the counter part of percepts, and the means for an agent to interact with the world. Generally actions defined in Prometheus maps to External Actions or ROSAPL Capability functions. They need implementation also in both layers. The difference is that the information flow start in the Cognitive layer and is passed to the Operative Layer. If the Action refers to a Capability, then it also implies the Executive Layer. Actions are steps inside an agent plan.

Snippet 2 illustrates how the *User Monitor* role could activate the *Get User Profile* inside the *Load Profile* plan. After the external Action is invoked the UserDBClient environment will route it to call the proper Java method that implements the real action in the environment. In this case access to the UserDB data and retrieve the user profile.

**Message**

Messages are treated as events, but have a special signature imposed by 2APL. The signature was already presented in Chapter 4, Section 4.2:

```
message(sender, performative, language, ontology, content)
```
In Java Environment the Percept event is thrown.

```java
/* The RosAgent has registered a capability that listens to
 * rostopic publishing if there is people in the
 * scene or not */
public SomeAwesomeCapability {
    ...
    private void newPersonsDetected(String personsName)
    {
        //event signature: newPerson(<peronsName>).
        APLFunction event = TermFactory.get("newPerson",personsName);
        //throw event throught environment
        getEnvironment().throwEvent(event,get_id);
    }
    ...
}
```

Then a module catches this event with a specific Procedural rule matching its signature.

```
pc-rules:
newPerson(P) <- not member(P,loggedUsers) |
{
    if (not G(identifyUser(P))
    {
        adopta(identifyUser(P);
    }
}
```

Snippet 1: Example of Percept implementation

To process incoming messages then, as well as with events, the developer needs to define Procedure Rules that match the specific signature of the message to process. Snippet 3 shows how the Robot could process the Accept and Reject messages inside Process CFP capability.

The first rule will enqueue the task as it was accepted by the contractor and adopt the goal to execute it. Rules for that goal will schedule the task accordingly with the capability integral logic and functionalities provided by its ROS API.

For internal messages inside an agent such as Generate Plan in Avatar agents we could define it as an Abstract Actions and therefore use the same mechanics as for processing external messages. This enforces the idea of relations and interfaces presented in Chapter 4, Section 4.4. For now, the devel-
5.5. **ROSAPl INHANDS IMPLEMENTATION**

**pg-rules:**

\[ \text{loadProfile}(P) \leftarrow \text{not userProfile}(\_ \_ ) \text{ and not activePlan}(\_) \ |
\{ \\
    @userDBClient(getUserProfile(P),PROFILE);
    loadP(PROFILE);
\}
\]

\[ \text{loadProfile}(P) \leftarrow \text{userProfile}(A,\_,\_ ) \text{ and not activePlan}(\_) \ |
\{ \\
    [ /* we want the unload operation to block any other plan, just in case */
    unloadUserProfile(A);
    if (B(\text{not userProfile}(A)))
    \\
    \\
    \{ \\
        @userDBClient(getUserProfile(P),PROFILE);
        loadP(PROFILE);
    \}
\]
\]

**pc-rules:**

\[ \text{loadProf}(Profile) \leftarrow \text{true} \ |
\{ \\
    if (B(PROFILE = \text{user}(\text{Name},\text{Type},\text{Reach\_Distance})))
    \\
    \{ \\
        +userProfile(\text{Name},\text{Type},\text{Reach\_Distance});
        dropa(loadProfile(P));
    \}
\}
\]

Snippet 2: Example of Percept implementation

oper is the one that needs to ensure that the interfaces is respected and only can associate two capabilities with inheritance in order to use abstract actions as interface. If for some reason it is not suitable to use module inheritance, he can use module instantiation provided by 2APL and set belief triggers to start the desired process in the child module.

snippet 4 illustrates a possible approach, using 2APL module Instances, on how to avoid importing the entire *Generate Plan Capability* inside Manage Recipes, or vice versa, in order to trigger plan generation when a recipe is loaded. Avoiding the need to have all the believes and possible rules that a plan generation possesses. This will improve the performance of the module if the generation of the plan is complex.
pc-rules:
message(S, inform, apapl, aplfunction, accep(PROPOSAL) <-
   pendingPropositions(L) and member(PROPOSAL,L) |
   { 
     if (B(PROPOSAL = proposal(Task,cost))
     { 
       UpdateExectuionTasks(Task);
       B(Task = task(ID,_,_,_));
       remove(PROPOSAL,L);
       +adoptz(execte(ID));
     }
   }
}
message(S, inform, apapl, aplfunction, reject(PROPOSAL) <-
   pendingPropositions(L) and member(PROPOSAL,L) |
   { 
     remove(PROPOSAL,L);
   }
}


/* Module Manage Recipes */
...
pc-rules:
selectedRecipe(R) <- true |
{ 
  @recipesDBClient(getRecipe(R),RecipePlan);
  create(GeneratePlan,plangenerator);
  plangenerator.updateBB(RecipePlan);
  plangenerator.execute(planGenerated);
  plangenerator.B(generatedPlan(PLAN));
  +assistivePlan(PLAN);
  adopta(MonitorRecipe);
}
...

Snippet 4: Internal Messaging using child modules and interface beliefs
5.6. EVALUATION

Protocol

A protocol could be defined as a system of digital rules for data exchange between agents. 2APL, as a rule based system, make protocol implementation very simple. In ROSAPL we proposed to define a set of generic modules that handles this rules. snippet 5 presents the abstract module that implements the Request protocol\(^1\). Then, the final system wherever is the need to use a particular protocol build upon the standard protocol the developer only needs to inherit from the mentioned abstract module and implement only how to set the initiation Goal and then define the abstract actions that manage the message succession. Snippet 5 illustrates the final implementation for the module that allow the Avatar to find Robots that provides needed capabilityes.

Capability and Agents

Capabilities in Prometheus are any functionality that an agent provides, generally by managing a reasoning process through percepts, actions and message protocols. So a capability is an aggregation of all the rules and modules that serve in one way or another to the main goals of the defined functionality. In general, it will be a module that later will be included or managed by the final top-level Agent modules. For further information on how to program them, the interested reader can refer to the 2APL User Guide\(^2\) or Chapter 4, Section 4.2, where ROSAPL Capabilities are described in detail.

5.6 Evaluation

The result of our implementation is a system that handles planning and task allocation for at user level. This allows the system to address at the same time the needs of users across different scenes(rooms) that could share the same resources. Furthermore execution control is split into each robot or capability provider device, avoiding the need of expensive central computational power and allow hight level of adaptation. Finally, if the communication protocols and interfaces are properly followed, it will make possible to add new functionalities to the system with not to much work, at least should not require modifications in other software components.

While we call the capability provider agent a Robot, that does not mean that a robot should be implemented by a single agent. But it is probably a good idea to maintain a single agent as a interface. This agent façade will then implement the necessary logic for negotiation and finally orchestrate the

/* Request Capability Provider */

... pc-rules:
%rule to start request
pendingTask(Task) <- DelegatedTask(L) and not member(Task,L) |
  {   request(provider(Task));%interface
      adopta(delegate(Task));
  }
%interface function to process request
processRequest(task(ID,action,_,_) <- true |
  {   adopta(generateListProviders(action,ID));
     B(listGenerated(L,ID));
     dropa(generateListProviders(action,ID);
     if(B(L=[]))
     {   failure();
     } else
        {   informDone(L,ID);
        }
  }
}

processFailure(ID) <- delegateTask(task(ID,_,_,_)) |
  {   noProviders(ID);
  }

processInformResult(L,ID) <- delegateTask(task(ID,_,_,_)) |
  {   callForProposals(L,ID);
  }
...

Snippet 5: Request Capability Providers module Implementation
execution of all the required capabilities having its agent. There is nothing against each capability or set of capabilities could have it’s own governing agent. For this thesis however we do not go this farther, as each robot has a couple of capabilities that a single agent should be capable of manage.
Chapter 6

Conclusions

In the introduction (Chapter 1) we presented the dual objective of this Master thesis. From one side to provide a design proposal that addresses the reasoning and task allocation for the prototype system the Robotics Group is developing at IBEC’s robotics lab. In the other hand to study the process of designing multi-robot systems and provide some methodological and technological basis upon someone could work on, as there was none found that approached the basic needs of such system.

In the following sections we analyse the results for each side and evaluate whether the proposed objectives were meet or not. Next we remark the general contributions of this thesis and finally point out possible future lines of work that could follow in order to improve the results or support other lines of research.

6.1 ROSAPL framework

Nowadays, integrating different robots to solve a task is not an easy. Most of them are not even designed to interact with sibling units. Besides there are many proposed frameworks to address social behaviours such as team formation, coordination or task allocation to mention a few; none of them contributes with a basic framework upon which start to build similar systems or use the proposed framework at implementation level, becoming ad-hoc solutions for a specific scenario or problem.

Building robots without social capabilities in a world that is more a more interconnected everyday is strange. There was indeed a gap between robot development and multi-robot development that requires to be implemented ad-hoc.

The ROSAPL framework seems a reasonable first step to fill this gap
in a world where robots left structured environments and are entering the unpredictable real world. In this new scenario social capabilities are a must, not only to interact with other units, but also to interact with humans and work arm to arm. While the ROSAPL framework presented in this thesis is still in its first stage of development, its functionalities are already more than enough to make robots of different kinds interoperate without much complexity overheat. Even probably help to reduce the complexity on how robots are programmed to act and use all their capabilities.

6.2 InHANDS

InHANDS was the perfect candidate project to develop our ideas and test their validity. It was a project that was facing the difficulty of integrating a few robots to assist people in the kitchen. Their initial approach was to build a central system that will govern all the robots and decide where each one of them should be used. This was requiring a tremendous design effort which was difficult to extend and maintain.

Once we suggested to look it from another point of view like agent-oriented solution, analysed the requirements and the future applications of their system, they agreed that the agent-oriented approach had some potential.

We designed InHANDS reasoning, coordination and execution systems following the ROSAPL ideas. We knew that the scenario was a challenging one and, due to time constraints, in this thesis we haven’t achieved a fully functional solution, but a proof-of-concept. Many functional requirements need some more work to be covered. In the other side, all the non-functional requirements are covered with ROSAPL together with the design and prototype presented. When presented, people at InHANDS agreed that, besides it was not complete, the approach and prototype is good enough to try to develop the final solution for its scenario using it.

6.3 Contributions

The technical contribution of this thesis is twofold. From one side we extend 2APL language to be used in high-level robot control applications. At the same time we add a few novel capabilities to ROS allowing robot to robot communications and easy robot behaviour implementation through a BDI approach. These are being prepared to be released soon as a ROS package that will available for the ROS community on Github\textsuperscript{1}.

\textsuperscript{1}ROSAPL - http://github.com/ROSAPL
Another important contribution of this thesis the methodological approach to implement real multi-robots systems based on BDI agents. This thesis also contributes with a basic framework for the implementation of such systems which is already going to be applied in a real system and be tested on the field in the context for the InHANDS project.

Although there has been no publications during the thesis lifetime, there are already plans to publish some of this thesis results on several international venues.

After hours of research and test, we believe that we achieved a feasible approach that could serve as a base to develop a functional framework for multi-robot systems. We also explored the possibilities of programming intelligent and social robots using BDI paradigm as a framework. BDI agents seem promising to power the mind of a robot in social setups, if the focus is made on the right level of abstraction.

6.4 Future work

ROSAPL put the bases to address the big problems of multi-robot systems. This have two consequences, the first, that derives to the most immediate lines of work are related to the refinement and evaluation of the framework. In the long and middle term the ROSAPL framework could be used for test and benchmark solutions of already proposed frameworks on some research areas such the following ones.

- Assistive Technologies with coordinated robotics and healthcare institutions.
- robot team formation
- collaborative and cooperative task execution.
- human-robot team work.
- task allocation in teams.
- coordination of robot teams.

But before go on that directions, the immediate work as far as framework, methodology and constructs in this thesis are concerned, the following extensions/improvements are possible.

Abstract Modules and Interfaces: There has been detected the need of using module abstract definition and interfaces to allow better organize
CHAPTER 6. CONCLUSIONS

the code structures and reasoning flows of the system. This will allow to switch between different approaches for the same task wherever its implementation modules are provided, relieving the developer for hand check the correct use of the designed interface and develop easily interchangeable modules.

**New Module Relations:** Improving the actual module system of 2APL to allow composition and association in addition to inheritance will allow to implement capability architectures such as the ones presented in Section 4.2. This will also enhance Abstract Modules and interfaces to better structure the code, the reasoning rules and plans.

**Integrate Shared Belief and Goals:** Upgrade the proposed 2APL extension in Dastani[CDH11] and integrate it in the current platform.

**Asynchronous Module Execution:** The actual modular system blocks all the agent processes when a child module is executed. This limits the kind of things an agent could do. Forcing to generate new agents that will be in charge of executing this functionality while we only pretend to divide some of the reasoning of the agent to the background to not affect the main agent process, while we pretend to share believes between the two modules in real time. This Asynchronous Execution and the integration of shared belief and goal interfaces already proposed by Dastani[CDH11] will really expand the 2APL possibilities.

**Module paths:** Allow to configure 2APL with folders where modules can be found. This will allow sharing modules, interfaces and abstract modules without have to keep all the used modules in the same folder as the MAS definition files.

**Add a project wizard generator** to help initial setup for basic system and help to introduce the framework and its architecture.

**Improve Rosjava integration** by providing useful scripts to generate custom messages, services and actions abstracting form the peculiarities of Rosjava client.

**Integrate simulation environments:** A weak point in the framework could be the absence of tools to integrate a simulation environment where to test the developed system in an easy and controlled manner. This is not actually a problem in the field as there are good projects that help to simulate robots. So a good step to make would be to add wrappers in a way that starting basic simulations don’t require to have a big understanding of them. For instance we could expand the explorer and carrier example distributed by 2APL platform to use one of the mentioned simulators in Sections ?? to integrate real-time robot simulation on the proposed multi-robot development cycle.
Provide a decentralized communication architecture that will allow to communicate agents from different networks thus avoiding the need to reside in the same LAN to communicate with other users, allowing, for instance, the use of 3G communications. Running the main Jade container as a standalone application will prevent the death of a agent or the main ROS system to kill the entire network. As stated before, there is no use on trying to achieve a distributed system where robots communicate by means of auto discovery systems. It will be interesting to extend 2apl messaging to be able to dynamically allocate its servers and implement this as a network service that can be for example moved to the cloud to make it more reliable, and resemble human messaging systems that nowadays are so popular. In this area could be interesting to see how ROSCON[SLK] approach could stand as a solution and how integrate with ROSAPL as network architecture where to run Jade (or any other messaging middleware) as a more dynamic agent systems.

Improve the methodology based on the implementation experience of Inhands and apply it in different scenarios to evaluate its generalization capability in order to improve the explanations and methodological approach, converting our current methodological sketch into a full methodology.

Integrate OWL reasoning and standard ontologies on communication protocols and languages to enhance the interactions between 2APL-based systems and other frameworks, basically JADE-based, as an example of how to integrate different communication layers in 2APL.

With Respect to the possible future work in the InHANDS project, the following aspect could be improved or considered:

Clinical Assessment would by large improve how the user profile is defined and which are the relevant aspects the system should focus on.

Activities of Daily Life(ADLs introduction could also extend the project to also attend people with cognitive disabilities helping them to follow specific schedules or treatment doses if they need so as some of the reviewed projects in Section 3.1.

Incorporate caregiver and professional roles to the system so it could log the users actions and performances in order to provide patient evolution and alerts where necessary to the caregivers so they can pass by and check if it is necessary.
Bibliography


[BCA] Toni Benedico, Carlo Caltagirone, and Roberta Annicchiarico. A SHARE-it service to elders mobility using the i-Walker.


[CACL] F Campana, R Annicchiarico, C Caltagirone, and Fondazione Santa Lucia. Towards an Intelligent Service to Elders Mobility Using the i-Walker.


Appendix A

ROSA PL

User Guide
introduction

The ROSAPL framework is developed to support the implementation and execution of multi-robot systems based on BDI\(^2\). The package allows robots based on ROS\(^3\) to be programmed using 2APL\(^4\) programming language in order to become part of a multi-agent system.

Essentially it is a ROS package that allows the execution of a 2APL interpreter as a ROS node, and as consequence, a MAS defined in a .2apl file. It also provides 2apl modules, Java classes and scripts in order to facilitate the development of a final multi-robot solutions.

In this user guide, we first explain how ROSAPL software can be downloaded, installed and used. In Chapter ?? we describes the basic steps to set a ROSAPL project and in Chapter ?? how to program a multi-robot system with it.

This guide presumres you are already familiar with ROS, it’s Java client Rosjava and 2APL language. If it is not the case refer to their beginners guide or documentation. For ROS start by following the main tutorials about ROS and Rosjava. For an introduction to 2APL, start from the official documentation.

Software Requirements

ROSAPL is a ROS package based in Rosjava. This theoretically should made it portable between windows, linux and Mac OS if they have Java Runtime Environment (JRE) 6 installed. We found out that due to a breaking change seems to not execute properly in newer versions newer than 1.6. While it is not required, it is recommended to have installed a Git version control system to manage the sources and dependencies of your project.

This User Guide focuses on develop under Linux environment so this are

\(^2\)Belief Desire and Intention
\(^3\)http://ROS.org
\(^4\)http://apapl.sourceforge.net
the requirements for ROSAPL under Linux Ubuntu 12.04 LTS. This is the requirements check-list before starting with ROSAPL.

- ROS Hydro
- Rosjava client and libraries.
- JRE 6 to use as default.
- 2APL_fork

ROS

ROSAPL is aimed to be used with ROS, so you should have some distribution to run it with. While theoretically Rosjava enables your nodes to be run in any machine that support JRE 6 running a roscore any part of ROSAPL was tested in other system than Linux Ubuntu 12.04 and ROS Hydro. Besides that, it should work in ROS Indigo too.

If you don’t have a running ROS distribution please follow the indications in http://wiki.ros.org/hydro/Installation to install it.

Once you have installed ROS you should make sure that have also installed the following packages:

- rosjava
- rosjava_msgs
- rosjava_build_tools

If you don’t have one, create a new catkin workspace where you will download ROSAPL project and, we suggest, implement your project.

Gettint 2APL interpreter

ROSAPL will not work with the standard distribution of 2APL. This distribution had some design limitations and bugs that only could be addressed modifying the source code, if we would like to keep it simple. For this reason we made a fork of the original 2APL platform developed at the Intelligent Systems Group of the Computer Science department of the University of Utrecht. All the changes we made so far could, eventually, be merged on the official repository. For now we made them public in a Github repository you can clone or fork:

http://github.com/ROSAPL/2APL_fork
List of Changes

This is the list of changes from the original release:

- 2APL can be launched and started from command line directly, without GUI, using Jade messaging layer.
- 2APL function has a static member to declare null functions (null(0)) to avoid some NullPointerException bugs.
- Environment class has his member protected so it can be properly subclasses and extended outside apapl package.
- APLNum is created using String as source. This is the recommended way to create BigDecimal instances and avoid precision issues that later make impossible to compare double values. Before, as example, 2.3 double was stored as a 2.2999999 BigDecimal value.

Installation

You can download the zip release that you can find in the repository. Alternatively you can clone the repository and build it from source execution the following commands.

```bash
>: cd /path
>: git clone http://github.com/ROSAPL/2APL_fork.git 2APL_fork
>: cd 2APL_fork
>: ant grandle
```

This will compile and package 2APL properly and set the proper path so rosapl can find and compile. If you downloaded the zip released extract it wherever it fells better to you. Then run the script `declare_grandle_path.bash` using as argument the path where you extracted the path. If you open a terminal in the extraction folder you could execute the following in the command line:

```bash
>: ./declare_grandle_path.bash $PWD
```

Getting ROSAPL

ROSAPL is a Rosjava package that is yet in its alpha version. Because of that it was not yet released as a official ROS package. To access the source code and use the package you must clone it from the official repository:
Clone it into the `src` directory in the ros workspace you will be using. If you properly setup 2APL_fork and Rosjava the only thing that remains is to build the workspace.

If you get an error telling you that Maven repository folder could not be created, you have forgotten to source the workspace setup.

```
ros_workspace_folder >: source devel/setup.bash
```

If everything is properly configured the build should not fail and you can run the example.

```
source devel/setup.bash
roslaunch rosapl legacy_interpreter.launch
```

This should show you the 2apl platform interface. Try to execute any example from 2apl_fork. This test runs only the original platform. Not ROSAPL MAS systems could be initialized with that. It is just for legacy test purposes.
ROSAPL Java Framework

ROSAPL goal is to aid developers to build multi-robot systems using an agent-oriented engineering approach. Agents are specially suitable to cope with the complexity of a system in which parts should play specific roles acting proactive and autonomously.

The framework integrates for you the 2apl interpreter. If you are not familiar with 2apl we recommend first to familiarize yourself with the language working out some of the examples they provide. As it is based on this platform, the way to define a mas is exactly the same. We only propose a pattern to use it for multiple robots integrating the same Multi-Agent System (MAS).

In addition to the basic 2apl functionalities, our framework delivers to the developers a set of classes to abstract them from ROS or 2apl internals. This is what it is presented in this chapter.

Figure ?? introduces you to all o the available constructs that will aid you to define agents to control a ROS powered robot. We dedicate a section to each of them to describe their use and functionality. As the API is not stable and it is in its first alpha release there are only documented the principal classes to be used or understand. For more detailed information refer to the API documentation in Github\(^5\)

If you find any big, or will like to suggest features or any other improvement feel free to use the issue tracker at GitHub repository

\(^5\)ROSAPL API: http://github.com/ROSAPL/rosapl.io

!h

Figure 1: ROSAPL base classes

101
RosEnvironmentBase

This is an abstract class that should be used as a base class for any environment you develop. It will allow to use all the functionalities that Rosjava provides to a node as well all the 2apl natural calls to throw events or activate external actions.

It also implements the logics to call a capability function, or which is the same, any public function in a class inheriting from Capability that is possessed by any agent implementing RosAgentBase.

To call that function from a 2apl external action use the following template in your module

@env(capability(capability_name,method(arg1,...,argn)),REST)

Where capability refers to the name you gave to that Capability when you added it to the RosAgent owned by the module.

RosAgentBase

This class will be probably the most used together with Capabilities (Section frame:cap). Its used to model each agent type, and even specific Agents of a Type with particular configurations. This is achieved by the Capabilities that are given to the agent. All methods of the registered Capabilities are accessible to 2apl modules as an external action thanks to the RosEnvironmentBase(Section 6.4).
AgentCapabilityBase

A Capability is an action or set of them that your agents knows how to perform and therefore access or modify the environment where your robot moves. One of their principal commitments is to give you access to ROS Topics and Services. Unfortunately Rosjava do not support Actions yet, if you try to access an action server you should define how to deal with it while we integrate them to our supported ROS APIs.

To implement capabilities there are some base classes that abstract you from how to interface them. Manage ROS interfaces with those base classes is a way lot easier than using raw Rosjava.

SensingCapabilityBase

Allows to publish to a topic. To create it only need to specify the topic name and type class.

SensingCapabilityBase

Subscribe a topic i a lot easier using this base class than raw Rosjava. You need to implement the Rosjava messageListener interface some where and use the method addMessageListener() of the capability to set it as callback. Every time a message arrives the messageListener callback will be executed.

ServiceServerCapabilityBase

Interface to Rosjava services, only need to provide the service name and type class. This base class is used to implement service provider.

ServiceClientCapabilityBase

Interface to Rosjava services, only need to provide the service name and type class. This base class is used to implement service provider.

TermFactory

2apl modules uses a reduced set of types derived from Term class in 2apl library. This types are difficult to understand at first, and transform from standard types requires a bit of code each time. To make the code more readable we provided with a TermFactory that will convert any supported standard type to a 2apl compatible representation. It will event convert collection of objects or types to the proper Term. Yo can even convert a class to something 2apl will know how to deal if such class implements APLObject interface.

---

6 see official documentation at: http://rosjava.github.io/rosjava_core/0.1.6
APLObject

Interface that forces to define how this object is represented in 2apl. Enables TermFactory to deal with your class and serialize it properly for 2apl interpreter. This will help to define for instance that the class ingredient should be seen in 2apl as ingredient(ingredient_name,quantity)

EnvState

Is a container to store the current state of the world, and this will automatically translate to 2apl as a fact in the form of:

\[
\text{state(state_id, [list of serialized content])}
\]

EnvState is an example of implementation of APLObject itself.

AgentMap

Is a container to store agents. RosEnvironment possess one by default. It can store different types of agents as long as they are implementing RosAgent interface. You may never need to invoke this class unless not using RosEnvironmentBase.
Execute a mas

Execute Rosjava nodes its simple, but the syntax is, at least, extrange. In addition, to configure the 2apl interpreter you will need to configure some ros parameters to do it because Rosjava not process roaparam arguments and not let you pass arguments to the final Java Main (in our case the Runner).

In orther to do so here we present a simple script that should serve as inspiration for your concrete case We suppose that the script is invoked using the Runner.launch.

```bash
#!/bin/bash

# $1 is the path is the path to the roapl package.
# r_path=$1/runner/build/install/runner/bin/runner
runner='com.github.rosapl.runner.Runner'

#load the necessary parameter to configure the #2apl containers that will be #launched in this #system
rosparam load config/config.yaml /apl
echo "configuration loaded."
echo "starting nodes."
gnome-terminal -e "$path $runner __name:=apl/container_00 "
#wait a bit until Jade comes up and main container is available
sleep 10
#wait until main container is ready!
gnome-terminal -e "$r_path $runner __name:=apl/container_01 "
gnome-terminal -e "$r_path $runner __name:=apl/container_02 "
gnome-terminal -e "$r_path $runner __name:=apl/container_03 "
echo "all containers where launched"
```

105
and here is the configuration file:

container_00: {
    masfile: /path_to_your/explorer_and_carrier/container00.mas,
    gui: true,
    jade: true,
    host: localhost }
container_01: {
    masfile: /path_to_your/explorer_and_carrier/container01.mas,
    gui: true,
    jade: true,
    host: localhost}
container_02: {
    masfile: /path_to_your/explorer_and_carrier/container02.mas,
    gui: false,
    jade: true,
    host: localhost }
container_03: {
    masfile: /path_to_your/explorer_and_carrier/container03.mas,
    gui: false,
    jade: true,
    host: localhost }
Appendix B

Inhands Design Proposal
System Specification

Goal Overview Diagram

Roles

System Role Overview Diagram
Environment Manager Role

Name: Environment Manager

Description: It is in charge of identifying the users, assign them to avatars and work as directory for robots.

Triggers: Request Register Capability, Request Capability Providers, User Enter Scene, User Left Scene, User Login

Actions: Create Avatar

Information used: UserDB

Information produced: Assigned Avatars, Free Avatars

Goals: Manage Environment, Identify User

Task Allocation Role

Name: Task Allocation

Description: When there are pending tasks, search for a suitable executor.

Triggers: Pending Task, Timeout

Actions

Information used: Pending Task,

Information produced: Delegated Tasks

Goals: Delegate Task

Capability Provider Role

Name: Capability Provider

Description: When receive a request, generates a proposal, if accepted executes a task

Triggers: Call For Proposals, Capability Success Capability Failure

Actions: Execute task

Information used: Capabilities

Information produced

Goals: Task Execution
Plan Monitor Role

Name: Plan Monitor
Description: Supervises the execution of the user plan, updating when delegated task finishes or user performs an identifiable action.
Triggers: User Action, Task Finished
Actions
Information used: User Plan, Delegated Task
Information produced: User Plan, Delegated Task
Goals: Monitor Recipe

Assistance Planner Role

Name: Assistance Planner
Description: Once the user plan is know, taking into account his profile will detect assistive actions, adapt their parameters and generate a list of Pending Task for the system.
Triggers: User Plan
Actions
Information used: User Plan
Information produced: Pending Task
Goals: Adapt Task, Determine Assistive actions.
**Scenarios**

**User Enters Scenario**

**Name**  User Enters

**Description**  User Appears in the scenario. System will ask for login. If user login it will assign an Avatar to him. If there is no Avatar, will create one.

**Trigger**  User Enters The Scene

**Steps**

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Percept</td>
<td>User Appears Scene</td>
<td>Scene Manager</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>Goal</td>
<td>Identify User</td>
<td>Scene Manager</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>Other</td>
<td>Wait for response</td>
<td>Scene Manager</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>Percept</td>
<td>User log-in</td>
<td>Scene Manager</td>
<td>User Login</td>
</tr>
<tr>
<td>(5)</td>
<td>Goal</td>
<td>Assign</td>
<td>Scene Manager</td>
<td>Assign Message</td>
</tr>
<tr>
<td>6</td>
<td>Goal</td>
<td>Load user profile</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Goal</td>
<td>Recover user state</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Goal</td>
<td>Receive user command</td>
<td>Avatar</td>
<td></td>
</tr>
</tbody>
</table>

**Variation**

In Step 3 User might not be login, the system awaits indefinitely for a user to login.
**User is cooking**

**Name**  
User is cooking

**Description**  
When Recipe Progress is active and use plan is generated, the system will start to monitor the user progress, it will ensure that the user has all ingredients and tools at its reach, and will dispose of everything else that was not expressly asked for by the user. The output of this scenario are assistive task identified, and updates in user plan.

**Trigger**  
Monitor Recipe Progress, User plan Loaded

**Steps**

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>Determine assistive task</td>
<td>Avatar</td>
<td>Pending Task</td>
</tr>
<tr>
<td>2</td>
<td>Goal</td>
<td>Needed object are in workspace</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Goal</td>
<td>Unneeded objects are not in workspace</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Other</td>
<td>Wait for user action</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Percept</td>
<td>User Action</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Action</td>
<td>Update user plan</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Other</td>
<td>Repeat from 6 until task finished</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Percepts</td>
<td>Plan completed</td>
<td>Avatar</td>
<td></td>
</tr>
</tbody>
</table>

**Variation**  
If user plan contains optional task that requires objects not in the work area ask which of those task will be performed.
Recipe Selected

**Name**  
Recipe Selected

**Description**  
When user is logged in it will be asked to select a recipe. User might or might not select a recipe. System will wait until user desires to select one. The result of this scenario is the generation load of a user plan

**Trigger**  
Goal: Select recipe

**Steps**

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Percept</td>
<td>User recipe query</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Action</td>
<td>Retrieve list of recipes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Goal</td>
<td>Select recipe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Other</td>
<td>Wait for response</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Percept</td>
<td>User recipe selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Goal</td>
<td>Generate user plan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Goal</td>
<td>Monitor recipe progress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Goal</td>
<td>Show recipe progress</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Variation**  
If user plan can not be generated, show the error to the user, and get back to step 2
Pending Assistance

**Name**  
Pending Assistance

**Description**  
Whenever a task where system can assist, it will be attempted to be executed.

**Trigger**  
Non empty list of task

**Steps**

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>Assign Task</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Goal</td>
<td>Execute Task</td>
<td>Robot</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Action</td>
<td>Execute capability</td>
<td>Robot</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Other</td>
<td>Wait End Execution</td>
<td>Robot</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Percept</td>
<td>Action Result</td>
<td>Robot</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Goal</td>
<td>Notify Result</td>
<td>Robot</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Other</td>
<td>Wait for result</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Percept</td>
<td>Action result</td>
<td>Avatar</td>
<td></td>
</tr>
</tbody>
</table>

**Variation**
User Task Request

Name: User Task Request
Description: A user assigned a task to the system. Who would need to execute it.
Trigger: User task query

Steps

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Percept</td>
<td>User Task Query</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Goal</td>
<td>Assign Task</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Goal</td>
<td>Respect user space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Goal</td>
<td>Execute Task</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Action</td>
<td>Execute capability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Other</td>
<td>Wait for result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Percept</td>
<td>Action result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Action</td>
<td>Show result to user</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Variation
User Leaves

**Name**  User Leaves

**Description**  There is no user in the scene, plan is finished. The system will clean the workspace and leave it ready for next session.

**Trigger**  No user detected and use plan finished. User logout

**Steps**

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Name</th>
<th>Agent</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Percept</td>
<td>User leave scene</td>
<td>Scene Manager</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Action</td>
<td>Unload user profile</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Goal</td>
<td>Store profile slot</td>
<td>Avatar</td>
<td>UserProfile UserDB</td>
</tr>
<tr>
<td>4</td>
<td>Goal</td>
<td>Notify Avatar Free</td>
<td>Avatar</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Message</td>
<td>Avatar is Free</td>
<td>Scene Manager</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Goal</td>
<td>Clean workspace</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Variation**

Step 3 could be optional if there is nothing to store.
**Percepts**

**Percept Selected Recipe**

<table>
<thead>
<tr>
<th>Name</th>
<th>Selected recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Happens when user selects a recipe in the interface.</td>
</tr>
<tr>
<td>Information carried</td>
<td>The ID of the recipe to use as a base plan</td>
</tr>
<tr>
<td>Knowledge updated</td>
<td>The actual recipe</td>
</tr>
<tr>
<td>Source</td>
<td>Environment</td>
</tr>
<tr>
<td>Processing</td>
<td>Load Recipe</td>
</tr>
<tr>
<td>Agents responding</td>
<td>Avatar</td>
</tr>
<tr>
<td>Expected frequency</td>
<td></td>
</tr>
</tbody>
</table>

**Percept Recipes Query**

<table>
<thead>
<tr>
<th>Name</th>
<th>Recipes Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A search to obtain a set of recipes</td>
</tr>
<tr>
<td>Information carried</td>
<td>Search parameter for recipes DB</td>
</tr>
<tr>
<td>Knowledge updated</td>
<td>Candidate List of recipes</td>
</tr>
<tr>
<td>Source</td>
<td>Environment</td>
</tr>
<tr>
<td>Processing</td>
<td>Select Recipes</td>
</tr>
<tr>
<td>Agents responding</td>
<td>Avatar</td>
</tr>
<tr>
<td>Expected frequency</td>
<td></td>
</tr>
</tbody>
</table>
### Percept User Action

<table>
<thead>
<tr>
<th>Name</th>
<th>User Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The last action user performed and that was identified</td>
</tr>
<tr>
<td>Information carried</td>
<td>name of action executed by user</td>
</tr>
<tr>
<td>Knowledge updated</td>
<td>User Plan</td>
</tr>
<tr>
<td>Source</td>
<td>Environment</td>
</tr>
<tr>
<td>Processing</td>
<td>Revise Plan</td>
</tr>
<tr>
<td>Agents responding</td>
<td>Avatar</td>
</tr>
<tr>
<td>Expected frequency</td>
<td>10hz</td>
</tr>
</tbody>
</table>

### Percept Time Out

<table>
<thead>
<tr>
<th>Name</th>
<th>Time Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Timer expired for await proposals</td>
</tr>
<tr>
<td>Information carried</td>
<td>CFP id that expired</td>
</tr>
<tr>
<td>Knowledge updated</td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>Environment</td>
</tr>
<tr>
<td>Processing</td>
<td>Task Delegation Capability</td>
</tr>
<tr>
<td>Agents responding</td>
<td>Avatar</td>
</tr>
<tr>
<td>Expected frequency</td>
<td>Once per CFP</td>
</tr>
</tbody>
</table>
### Percept User Login

<table>
<thead>
<tr>
<th>Name</th>
<th>User Login</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Credentials for user login</td>
</tr>
<tr>
<td>Information carried</td>
<td>User Id and confirmation</td>
</tr>
<tr>
<td>Knowledge updated</td>
<td>Logged User</td>
</tr>
<tr>
<td>Source</td>
<td>Environment</td>
</tr>
<tr>
<td>Processing</td>
<td>Manage Avatars</td>
</tr>
<tr>
<td>Agents responding</td>
<td>Scene Manager</td>
</tr>
<tr>
<td>Expected frequency</td>
<td></td>
</tr>
</tbody>
</table>

### Percept User Enters Scene

<table>
<thead>
<tr>
<th>Name</th>
<th>User Enters Scene</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A person appeared in the scene</td>
</tr>
<tr>
<td>Information carried</td>
<td></td>
</tr>
<tr>
<td>Knowledge updated</td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>Environment</td>
</tr>
<tr>
<td>Processing</td>
<td>Manage Avatars</td>
</tr>
<tr>
<td>Agents responding</td>
<td>Scene Manager</td>
</tr>
<tr>
<td>Expected frequency</td>
<td></td>
</tr>
</tbody>
</table>
### Percept User Left Scene

<table>
<thead>
<tr>
<th>Name</th>
<th>User Left Scene</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>A person dessapeared from the scene</td>
</tr>
<tr>
<td><strong>Information carried</strong></td>
<td>id of person desapeared</td>
</tr>
<tr>
<td><strong>Knowledge updated</strong></td>
<td>logged users, assigned avatars, free avatars</td>
</tr>
<tr>
<td><strong>Source</strong></td>
<td>Environment</td>
</tr>
<tr>
<td><strong>Processing</strong></td>
<td>Manage Avatars</td>
</tr>
<tr>
<td><strong>Agents responding</strong></td>
<td>Scene Manager</td>
</tr>
<tr>
<td><strong>Expected frequency</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Percept Capability Failure

<table>
<thead>
<tr>
<th>Name</th>
<th>Capability Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Were a robot capability fails, this percepts is throw</td>
</tr>
<tr>
<td><strong>Information carried</strong></td>
<td>The reason or state of failure</td>
</tr>
<tr>
<td><strong>Knowledge updated</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Source</strong></td>
<td>Environment</td>
</tr>
<tr>
<td><strong>Processing</strong></td>
<td>Robot</td>
</tr>
<tr>
<td><strong>Agents responding</strong></td>
<td>Robot</td>
</tr>
<tr>
<td><strong>Expected frequency</strong></td>
<td>Once per execution if fail</td>
</tr>
</tbody>
</table>
**Percept Capability Success**

<table>
<thead>
<tr>
<th>Name</th>
<th>Capability Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The execution of a capability was a success</td>
</tr>
<tr>
<td>Information carried</td>
<td>Result of the capability if any</td>
</tr>
<tr>
<td>Knowledge updated</td>
<td>Task result</td>
</tr>
<tr>
<td>Source</td>
<td>Environment</td>
</tr>
<tr>
<td>Processing</td>
<td>Robot</td>
</tr>
<tr>
<td>Agents responding</td>
<td>Robot</td>
</tr>
<tr>
<td>Expected frequency</td>
<td>Once per execution if success</td>
</tr>
</tbody>
</table>
### Actions

#### Execute Task

<table>
<thead>
<tr>
<th>Name</th>
<th>Execute Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Starts execution of a robot capability</td>
</tr>
<tr>
<td>Parameters</td>
<td>arguments for the invoked capability</td>
</tr>
<tr>
<td>Duration</td>
<td>Until it finishes</td>
</tr>
<tr>
<td>Failure</td>
<td>Error from ROS node</td>
</tr>
<tr>
<td>Partial Change</td>
<td></td>
</tr>
<tr>
<td>Side Effects</td>
<td>Modify the environment according to action</td>
</tr>
</tbody>
</table>

#### Get Recipe

<table>
<thead>
<tr>
<th>Name</th>
<th>Get Recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Query recipeDB for a specific recipe</td>
</tr>
<tr>
<td>Parameters</td>
<td>recipe ID</td>
</tr>
<tr>
<td>Duration</td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>return none</td>
</tr>
<tr>
<td>Partial Change</td>
<td></td>
</tr>
<tr>
<td>Side Effects</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Get User Profile</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>Description</td>
<td>Queries User DB for a specific profile</td>
</tr>
<tr>
<td>Parameters</td>
<td>User ID</td>
</tr>
<tr>
<td>Duration</td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>return none</td>
</tr>
<tr>
<td>Partial Change</td>
<td></td>
</tr>
<tr>
<td>Side Effects</td>
<td>actions can not be adapted</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Search Recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Query recipe DB for a subset of recipes</td>
</tr>
<tr>
<td>Parameters</td>
<td>search parameters</td>
</tr>
<tr>
<td>Duration</td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>return none</td>
</tr>
<tr>
<td>Partial Change</td>
<td></td>
</tr>
<tr>
<td>Side Effects</td>
<td>recipe is not loaded and process is aborted</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Update User Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Stores the state of the user in the userDB</td>
</tr>
<tr>
<td>Parameters</td>
<td>user profile</td>
</tr>
<tr>
<td>Duration</td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>return none</td>
</tr>
<tr>
<td>Partial Change</td>
<td></td>
</tr>
<tr>
<td>Side Effects</td>
<td>user profile is not safe, avatar is not free</td>
</tr>
</tbody>
</table>
Data

Data Coupling Overview:
### Assistive Plan Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Assistive Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Plan that contains all the possible assistive actions that system coils execute given the current scenario</td>
</tr>
<tr>
<td>Data Type</td>
<td>Plan</td>
</tr>
<tr>
<td>Included fields/aspects</td>
<td>Tasks, Task Relations</td>
</tr>
<tr>
<td>Persistent</td>
<td>No</td>
</tr>
<tr>
<td>External to system</td>
<td>No</td>
</tr>
<tr>
<td>Produced by</td>
<td>Assistive Planner</td>
</tr>
<tr>
<td>Used by</td>
<td>Plan Monitor</td>
</tr>
<tr>
<td>Used when</td>
<td>Generated when user is in scene while doing an activity.</td>
</tr>
</tbody>
</table>

### Capabilities Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Information about the capabilities a robot posses. This allows the robot to accept request for that capabilities, and the system know who can perform that capability</td>
</tr>
<tr>
<td>Data Type</td>
<td>List</td>
</tr>
<tr>
<td>Included fields/aspects</td>
<td>Capability</td>
</tr>
<tr>
<td>Persistent</td>
<td>Yes, in agent code</td>
</tr>
<tr>
<td>External to system</td>
<td>Yes</td>
</tr>
<tr>
<td>Produced by</td>
<td>Robot</td>
</tr>
<tr>
<td>Used by</td>
<td>Robot</td>
</tr>
<tr>
<td>Used when</td>
<td>Robot starts, to register capabilities in the system.</td>
</tr>
</tbody>
</table>
### Capability Providers Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Capability Providers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>List of capabilities and the agents who provide them.</td>
</tr>
<tr>
<td><strong>Data Type</strong></td>
<td>List</td>
</tr>
<tr>
<td><strong>Included fields/aspects</strong></td>
<td>Capability, Robot ID</td>
</tr>
<tr>
<td><strong>Persistent</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>External to system</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>Produced by</strong></td>
<td>Scene Manger</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Scene Manger</td>
</tr>
<tr>
<td><strong>Used when</strong></td>
<td>When Avatar ask for capability providers</td>
</tr>
</tbody>
</table>

### Delegated Tasks Data

<table>
<thead>
<tr>
<th>Name</th>
<th>Delegated Task</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>List of task that Avatar already delegated to Robots.</td>
</tr>
<tr>
<td><strong>Data Type</strong></td>
<td>List</td>
</tr>
<tr>
<td><strong>Included fields/aspects</strong></td>
<td>Task, ID of the robot to which was delegated</td>
</tr>
<tr>
<td><strong>Persistent</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>External to system</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>Produced by</strong></td>
<td>Avatar</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Avatar</td>
</tr>
<tr>
<td><strong>Used when</strong></td>
<td>Monitoring Plan</td>
</tr>
</tbody>
</table>
Free Avatars Data

Name               Free Avatars
Description        List of existing Avatars that have no user assigned
Data Type          List
Included fields/aspects Avatar ID
Persistent         No
External to system No
Produced by        Scene Manager
Used by            Scene Manager
Used when          A new user appears and need to be assigned after he has logged in.

Logged Users Data

Name               Logged Users
Description        List of users that have logged in and the Avatar to which each one is assigned.
Data Type          List
Included fields/aspects assigned(AvatarID, UserID)
Persistent         No
External to system No
Produced by        Scene Manager
Used by            Scene Manager
Used when          A user disappears from the scene, to release the avatar.
Object DB Data

Name: Objects DB
Description: Objects database provided by inhands scene
Data Type: Database
Included fields/aspects: Objects, properties, positions, volume
Persistent: Yes
External to system: Yes
Produced by: InHands
Used by: Avatar, Robot
Used when: Avatar planning and Robot task Execution.

Pending Proposals Data

Name: Pending Proposals
Description: List of proposals that a robot has made and not received a response yet
Data Type: Propose message
Included fields/aspects: Proposal, Receiver, Time
Persistent: No
External to system: No
Produced by: Robot
Used by: Robot
Used when: A response to a proposition arrives, this match the Data in one of the entries of this Data objects and when processed its deleted.
Recipes DB Data

Name: Recipes DB
Description: Database of recipes that the system knows and that may be what the user will want to prepare.
Data Type: Plan
Included Fields/Aspects: Tasks, Task Relations
Persistent: Yes
External to System: Yes
Produced by: InHANDS
Used by: Avatar
Used when: When the user logs in and selects what they want to prepare.

Tasks In Execution

Name: Task In Execution
Description: List of tasks the robot is actually executing
Data Type: List
Included Fields/Aspects: Task
Persistent: No
External to System: No
Produced by: Robot
Used by: Robot
Used when: Robot receives a result task percept, to process and inform appropriately to the contractor.
**User Profile**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>User Profile</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Describe the type of user, its level of paralysis, the side parallelized and its range of action</td>
</tr>
<tr>
<td><strong>Data Type</strong></td>
<td>Profile</td>
</tr>
<tr>
<td><strong>Included fields/aspects</strong></td>
<td>UserID, paralyzedSide, paralysisLevel, rageOfAction</td>
</tr>
<tr>
<td><strong>Persistent</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>External to system</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Produced by</strong></td>
<td>InHANDS</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Avatar</td>
</tr>
<tr>
<td><strong>Used when</strong></td>
<td>Planning, replan and task adaptation</td>
</tr>
</tbody>
</table>
Architectural Design

System Overview Diagram
**Agents**

**Robot**

- **Name**: Robot
- **Description**: Represent an actuator with assistive capabilities
- **Cardinality minimum**: 0
- **Cardinality maximum**: N
- **Lifetime**: As long it is connected
- **Initialization**: Loads all its capabilities
- **Percepts**: Capability Success, Capability Failure
- **Actions**: Execute Actions
- **Uses data**: Capability
- **Produces data**: Delegated Task, Task in execution
- **Internal data**: Delegated Task, Task in execution
- **Goals**: Execute Assistive Task
- **Functionalities**: Task Executor
- **Protocols**: Register Capability, Task Delegation Contract Net
Scene Manager

Name  Scene Manager
Description  Manages general environment features, identifies users and assign them to avatar. Serves as Robot Directory
Cardinality minimum  1
Cardinality maximum  1
Lifetime
Initialization  As system starts
Demise  As systems goes down
Percepts  User Enter Scene, User Left Scene, User Login
Actions  Create Avatar
Uses data  UserDB
Produces data  Capability Providers
Internal data
Goals  Manage Environment, Identify user
Functionalities  Manage Environment
Protocols  Register Capability Provider, Request Capability Provider, Assign Avatar, Release Avatar.
Avatar

Name  Avatar
Description  Represents the user, its intentions and generate assistance plans
Cardinality minimum  0
Cardinality maximum  1 per user logged in
Lifetime  While user is in scene
Initialization  When user appear in scene, loads its profile
Demise  When user leaves the scene
Percepts  Selected Recipe, Recipes Query, User Action, Time out
Actions  Search Recipe, Get Recipe
Uses data  Objects DB, Recipes DB
Produces data
Internal data  User Profile, Delegated Task
Goals  Personalize Experience, Prepare Recipe, Determine Assistive Task, select Recipe, Monitor Recipe
Functionalities  Assistance Planner, User Monitor, Plan Monitor, Task Allocation
Protocols  Find Capability Provider Contract Net
Capabilities

Execute Capability

Name: Capability Failure
Description: It is in charge of execute a capability in the robot.
Goals: Assist User,

Notes: Enqueue capability will be specific to each capability and the restrictions ROS API imposes for that specific capability. So a new task could mean the cancel of the previous one.
Monitor Capability

<table>
<thead>
<tr>
<th>Name</th>
<th>Monitor Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>When a robot capability is in execution when it finishes this capability catch the result.</td>
</tr>
<tr>
<td>Goals</td>
<td>Monitor Plan</td>
</tr>
</tbody>
</table>

![Diagram of Monitor Capability]

Notes

---

Lookup Capability Provider

<table>
<thead>
<tr>
<th>Name</th>
<th>Lookup Capability Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Generates a list of Robots for a concrete task to be delegated and send it back to the initiator</td>
</tr>
<tr>
<td>Goals</td>
<td>Delegate Task, Assist User</td>
</tr>
</tbody>
</table>

![Diagram of Lookup Capability Provider]

Notes
Manage Avatars

Name: Manage Avatars
Description: Maintains the list of Free Avatars and Logged Users when a user enters or leaves the scene
Goals: Personalize Assistance, Identify User

Manage Recipes

Name: Manage Recipes
Description: Search and retrieve Recipe Plans
Goals: Assist User, Personalize Assistance, Select Recipe
Manage User Profile

Name: Manage User Profile
Description: Accesses UserDB and retrieves User Profiles, it also saves persistent information about the user (logs, history...)
Goals: Personalize Assistance, Identify User

Generate Plan

Name: Generate Plan
Description: Generates a plan when it receives the request to do so, and returns it.
Goals: Adapts Assistance, Assist User
Monitor Plan

Name: Monitor Plan
Description: When there are existing plans, they are monitored by this capability, updating the progress when delegated actions finish and report back or the user actions match one actionable task in the plan. If there are unexpected events or a delegated task fails, it may trigger a replan.
Goals: Monitor Recipe, Assist User

Register Capability Provider Capability

Name: Register Capability Provider
Description: When a robot requests to register a Capability, it is registered
Goals: Delegate Task
Process CFP

Name: Process CFP
Description: When a CFP arrives to a Robot it is handled by this capability
Goals: Assist User, Delegate Task

Notes: If a reject arrives, the Robot deletes it from pending proposals and drops the query.

Register Capability

Name: Register Capability
Description: When Robot is created, the first thing that will be doing is register its capabilities
Goals: Assist user, Delegate Task

Notes
Task Delegation Capability

Name: Task Delegation Capability

Description: Wherever there are pending tasks, it will try to allocate them to a robot using a Contract Net Protocol.

Goals: Assist User, Delegate Task
Protocols

Find Capability Providers

<table>
<thead>
<tr>
<th>Name</th>
<th>Find Capability Providers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Interaction to obtain a list of providers for a capability that should be delegated.</td>
</tr>
<tr>
<td>Included messages</td>
<td>Request Capability Providers, Response Capability Providers</td>
</tr>
<tr>
<td>Scenarios</td>
<td>Pending Assistance, User Task Request.</td>
</tr>
<tr>
<td>Agents</td>
<td>Avatar, Scene Manager</td>
</tr>
</tbody>
</table>

Register Provider

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Interaction that a Robot starts to register itself as provider for its capabilities.</td>
</tr>
<tr>
<td>Included messages</td>
<td>Register, Inform</td>
</tr>
<tr>
<td>Scenarios</td>
<td>Any</td>
</tr>
<tr>
<td>Agents</td>
<td>Robots, Scene Manager</td>
</tr>
<tr>
<td>Notes</td>
<td>Inform tells robot that capability was registered or not.</td>
</tr>
</tbody>
</table>

Assign Avatar

<table>
<thead>
<tr>
<th>Name</th>
<th>Assign Avatar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Scene Manager assigns a user to an avatar so user can interact with the system as another agent.</td>
</tr>
<tr>
<td>Included messages</td>
<td>Assign Avatar</td>
</tr>
<tr>
<td>Scenarios</td>
<td>User Enters Scene</td>
</tr>
<tr>
<td>Agents</td>
<td>Scene Manager, Avatar</td>
</tr>
<tr>
<td>Notes</td>
<td></td>
</tr>
</tbody>
</table>
Task Delegation Contract Net

Name: Task Delegation Contract Net
Description: Interaction around a task that should be delegated
Included messages: CFP, propose, Accept-Proposal, Reject-Proposal, refuse, inform-done, failure, inform-result
Scenarios: User Task Request Scenario, Pending assistance Scenario
Agents: Avatar, Robot
Notes:

Diagram:

- Initiator
- Participant
- CFP
- Propose
- Refuse
- Accept-Proposal
- Reject-Proposal
- Failure
- Inform-done
- Inform-result
### Assign Avatar

<table>
<thead>
<tr>
<th>Name</th>
<th>Release Avatar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Scene Manager tells an avatar to be released when its assigned user leave the scene.</td>
</tr>
<tr>
<td>Included messages</td>
<td>Release Avatar</td>
</tr>
<tr>
<td>Scenarios</td>
<td>User leaves Scene</td>
</tr>
<tr>
<td>Agents</td>
<td>Scene Manager, Avatar</td>
</tr>
<tr>
<td>Notes</td>
<td></td>
</tr>
</tbody>
</table>