RUBICON
Robotic UBUquitous COgnitive Network

Project No.: 269914

D3.1 – Control Layer Architecture

Editor(s):	**Mauro Dragone**	**NUID UCD**
	Mathias Broxvall	**ORU**
	Federico Pecora	**ORU**
	Alessandro Saffiotti	**ORU**
	David Swords	**NUID UCD**
	Sameh Abdel-Naby	**NUID UCD**
Contributor(s):	**Claudio Vario**	**CNR**
	Claudio Gennaro	**CNR**

Dissemination level	
X	**PU** = Public
	PP = Restricted to other programme participants (including the Commission Services)
	RE = Restricted to a group specified by the consortium (including the Commission Services)
	CO = Confidential, only for members of the consortium (including the Commission Services)

Issue Date	30/09/2011 (M9, MS1)
Deliverable Number	D3.1
WP	WP3 - Control Layer
Status	☐Draft ☐Working ☒Released ☐Delivered to EC ☐Approved by EC

Document history			
V	Date	Author	Description
	30/09/2011	Mauro Dragone	First deliverable version for internal revision and quality assurance review
	14/10/2011	Mauro Dragone	Revised version after quality assurance review. Changed status to “Released”.
	14/10/2011	Ciaran Clissmann	Final review

Disclaimer

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

The document reflects only the author’s views and the Community is not liable for any use that may be made of the information contained therein.

Executive Summary

This deliverable (D3.1) takes place at the end of the first task (T3.1 - Functional Design and Specifications) of WP3 - the RUBICON Control Layer. The goal of the Control Layer is to exercise high level control over the sensing/acting/communication capabilities of the RUBICON system. This control focuses on devising suitable action and configuration strategies while exploiting the RUBICON Learning Layer (WP2) to adapt these strategies to the environment and improve their quality over time.

Together with D1.1, D2.1, and D4.1, this deliverable presents a set of requirements and specifications to support the development of the RUBICON system .

The requirements reported here provide a collection of statements to inform research directions for the RUBICON project. They are based on several inputs: the Description of Work (DoW) document, a Closed Workshop (June 20-23, Örebro), the case studies described in Section 2.1 (which combine contributions from all the RUBICON partners), the requirements described in deliverables D2.1 and D4.1, as well as input from several informal discussions among the project consortium.

These inputs are used to carry out domain analysis, leading to the identification of the requirements for the Control Layer. After that, this document examines the state of the art in control solutions applicable to robotic ecologies in order to provide a first approximation of the design of the control layer and its most important interactions with the other layers of the RUBICON architecture.

Finally, this deliverable provides the specifications of a test-plan to be used for the development and the evaluation of the various releases of the RUBICON middleware.

Contents

Abbreviations

RUBICON	Robotic UBIquitous COgnitive Network
PEIS	Physically Embedded Intelligent System
BDI	Belief Desire Intention (agent model)
AF	Agent Factory

Figures

Tables

1. Introduction

1.1 Overview of the High-Level RUBICON Architecture

This project will create a self-sustaining, self-organizing, learning and goal-oriented **robotic ecology**, called RUBICON (Robotic UBIquitous COgnitive Network), where a robotic ecology is defined as a network of heterogeneous computational nodes interfaced with sensors, effectors and mobile robot devices.

The nodes of a RUBICON ecology mutually support one another's learning. RUBICON seeks to deliver learning solutions yielding cheaper, more adaptive and more efficient configuration and coordination of robotic ecologies, in support of open, dynamic, heterogeneous and computationally constrained systems, as well as a wide range of services and end user applications.

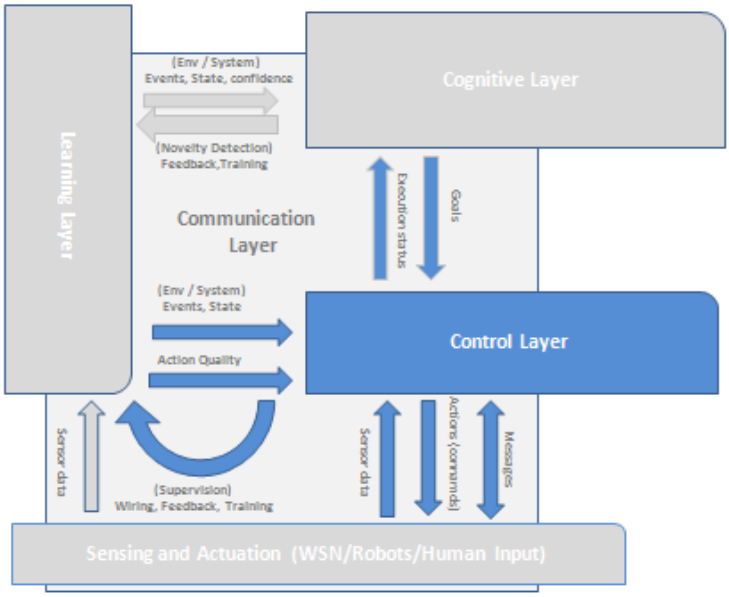

Figure 1. The RUBICON High Level Architecture (Control Layer highlighted).

Figure 1 depicts the conceptual, layered high-level architecture of the RUBICON system, outlining the main responsibilities of each layer and their main interactions and emphasising the Control Layer.

- The **Communication Layer** provides inter-component **communication** and integration mechanisms by leveraging and extending state of the art solutions in WSNs and middleware for robotic ecologies.

- The **Learning Layer** provides a distributed, adaptive, and self-organizing **memory** comprising learning neurons residing on multiple nodes of the RUBICON ecology. These neurons interact and cooperate through the underlying communication channels provided by the Communication Layer.

- The **Control Layer** provides **high level control** over the nodes of the RUBICON ecology, by formulating and executing both action and configuration strategies to satisfy the objectives set for the RUBICON ecology and the necessary collaborations for each node. The Control Layer uses the Learning Layer to refine the perception capabilities of the RUBICON ecology and to adapt action and configuration strategies to the environment.

- The **Cognitive Layer** provides **cognitive reasoning** methods to analyze the situation, **set the goals** of the RUBICON ecology, and exploit robot mobility to satisfy application requirements, gather knowledge, and drive the self-learning capabilities of the RUBICON ecology to increase its performance over time.

1.2 Overview of the RUBICON Control Layer

Figure 1 lists all the inter-layer dependencies resulting from the requirement analysis and specification activities jointly carried out in D1.1, D2.1, D3.1 and D4.1.

The main objectives of this layer are:

- formulating and executing both action and configuration strategies to satisfy the goals set by the Cognitive Layer and the necessary collaborations set for each node.
- exploiting the RUBICON Learning Layer to improve its perception capabilities and adapt these strategies to the perceived situation of the environment and the system, and to improve the quality of the same strategies over time.
- putting in place a vocabulary that clearly describes the capabilities of each component, and introspection and coordination capabilities to carry out actions and configuration plans with heterogeneous and open (with components leaving and joining the system) scenarios.

2. Requirements

The requirements analysis begins by reporting the functional requirements associated with the interfaces that the Control Layer must provide to other RUBICON layers. These requirements are then examined, together with the information in the DoW, to infer other non-functional and functional requirements describing the desired behaviour of the Control Layer. Finally this section examines the requirements requested of the other layers of the RUBICON architecture.

Requirements that are requested by a workpackage with number *X* to another workpackage *Y* are named *RX->Y.Symbol,* where *Symbol* is a short name recalling what flow of data, control, or functionality is associated to the requirement (e.g. STATUS, SCALABILITY). Note that there might be multiple workpackages requesting the same functionality: in this case, the associated numbers are separated by a comma, e.g. *RX,Z->Y.Symbol*. Requirements that are internal to the Control Layer are denoted as *R3.Symbol.*

2.1 Case Studies

This section describes a set of case studies that are used to illustrate the Control Layer requirements in practical application scenarios.

In order to focus on the key innovations tackled by the project, case studies have been modelled on the ambient assisted living (AAL) and transport applications to be developed in WP5.

Some of these case studies are also listed in deliverables D2.1 and D4.1 and are replicated here only for reference purposes. Rather than describing actual application scenarios, which will need to be designed and planned in WP5, these case studies are used to synchronize the requirement analysis carried out in all the technical workpackages, and to build a common understanding of RUBICON's capabilities.

The emphasis of these case studies is to show the RUBICON's ability to improve the quality of the services it delivers (e.g. (transport, activity recognition) over time, by coordinating its heterogeneous components and by leveraging its autonomous, self-sustaining learning capabilities to acquire, use, and share situated knowledge.

Table I - Case Studies in AAL Scenario

Name	DESCRIPTION
COOKING	The user prepares food once, twice or three times a day, usually in the morning, early afternoon and evening. The user's cooking habits may change, for instance, depending on the day of the week (working day/weekend) and the seasons of the year. However, before the meal is ready, the user rarely leaves the kitchen for more than few minutes. The RUBICON learns to recognize when the user is ready to prepare food. From the fact that the user usually switches on the extractor fan while cooking, the RUBICON will learn to automatically activate it when appropriate.

EATING	After cooking, the user usually eats his/her own meal after turn on the preferred combination/intensity of lights where he/she eats. This usually occurs either in the kitchen or in the living room, depending on the day of the week, or whether the user is alone or in the company of other people. At lunch time the user usually prefers the living room, where he/she can watch the news. The user usually activates the air refresher after cleaning the table. The RUBICON learns to recognize the user's habits and preferences, and will automatically turn on the ambient lights in the proper room when the user is getting ready to eat his/her meal, switch on the TV in the living room at lunch time, and activate the air refresher when the user has finished eating.
FIRE WARNING	The user starts cooking but then leaves the kitchen to answer a call to the main door where he/she engages in a lengthy conversation with the neighbour. The RUBICON learns to recognise this event as anomalous and reminds the user of his/her cooking in the kitchen.
FIRE ALARM	Occasionally, the user forgets the pan on the hob. The food burns and the resulting smoke triggers the fire alarm. One day, the batteries in the fire alarm run out. The RUBICON detects the smoke and recognizes the failure of the fire alarm when it does not ring. A robot seeks the user and advises him to switch off the hob and to check the batteries of the fire alarm.
WATER TAP	The user usually uses the tap in the bathroom sink and closes it before leaving the bathroom. The RUBICON learns to recognize when the tap has been left running,
BROKEN MICROPHONE	The microphone (the main sensor used to recognize the sound of water running from the tap) located near the bathroom's sink is muted. The RUBICON ecology recognizes that one of the normal events, a receipt of the sound event for the water tap, is not received although the user have entered the bathroom. This abnormal situation can be explained by two hypothesis, that there is no sound or that the microphone is broken. To verify the integrity of the microphone sensor the RUBICON employs an exploration task to sends a robot carrying a microphone to the bathroom to verify the lack of a sound event. It will thus confirm that there is a problem with the microphone when receiving diverging sensor readings from the static microphone and from the robot based microphone.
BROKEN MICROPHONES WITH RECHARGING ROBOTS	The microphone (the main sensor used to recognize the sound of water running from the tap) located near the bathroom's sink breaks. Robots cannot be used as they are all recharging their batteries. The RUBICON advise the user to check the bathroom and the microphone.
CLEANING	The user usually activates the vacuum cleaner in the kitchen after washing the dishes or in any of the rooms whenever he/she feels that the room in question is too dusty. The user usually avoids remaining in the same room while the vacuum cleaner is on, and stops the cleaner if he/she needs to be in

	the same room. The RUBICON learns to activate and to send the vacuum cleaner to clean the most appropriate rooms, and to suspend it whenever the user walks into the same room.
EXERCISE BICYCLE	The user buys an exercise bicycle. He/she places the bicycle in the corridor, and starts to use it every second day of the week, usually before dinner. The RUBICON learns to recognize the user's new activity and to predict when the user is ready to exercise.
ALARM WHEN CLEANING	The vacuum cleaner is currently cleaning the living room while the user is away. The microphones in the kitchen detects a noise but the RUBICON is not sure about its cause, as it is disturbed by the noise caused by the cleaning. The RUBICON learns to send the mobile robot with a microphone (or a camera) to check the kitchen. Additionally, the RUBICON learns to temporarily suspend the cleaning in order to reduce the background noise and maximize the chances to understand what's happening in the kitchen.
KITCHEN WALL	The user install a light wall to separate the cooking area from the rest of the kitchen. As a result, the RUBICON now finds difficult to recognize when the user is cooking, as the microphones and other sensors were first installed far from the cooking area - out of reach of dangerous smoke and vapours. The robot starts to go in the kitchen when the user is also there in order to recognize the user's activities. After some time, the RUBICON learns to recognize the same activities with the sensors already installed in the kitchen and the robot ceases to always follow the user in the kitchen.
LIGHTS	The user usually switches on the lights as soon as he/she walks into a dark room, and switches them off just before he/she leaves it. The RUBICON learns to switch on the lights in every dark room the user is about to enter, and to switch them off as soon as the user has left.
PLANT WATERING	The user usually waters the plants with a watering can filled from the tap in the kitchen. After watering, the user prefers to keep the room in low light for some time. The RUBICON learns to close the blinds and/or to switch off the lights if there is too much light after the user has watered the plants.
PLANT WATERING ON CLOUDY DAY	On a very cloudy day, the RUBICON does not close the blinds after the user has watered the plants. After a while, the RUBICON closes the blinds as the day brightens up and lots of light enters into the room
NIGHT	The user usually goes to sleep in the evening after making sure that all the appliances (e.g. TV, radio, vacuum cleaner) and lights are off, the blinds in the bedroom are down, the water taps are closed, the main door and all the windows are locked, and the burglar alarm system is activated. The RUBICON learns to switch off all the appliances, activate the alarm system and inform the user about anything that may need attention if it cannot be operated automatically or by a robot.
NAVIGATION - CARPET	The user buys a carpet and places it at the entrance of the bathroom. After a months, the user decides to remove the carpet.

	The RUBICON learns to prefer to send the Nao whenever it needs a help in the bathroom, as the wheeled robot finds too difficult to navigate on the new carpet. After the carpet is removed, the RUBICON learns it can use again the wheeled robot, if it needs to.
RSSI-BASED ROOM LOCALIZATION	Every time a robot navigates in a room with camera-based localization system, the RUBICON knows in which room the robot is located. The robot also senses the signal strengths of the various WSN nodes installed in the environment. The RUBICON learns to recognize in which room is the robot by just looking at the signal strengths information. If the user starts wearing a bracelet equipped with a WSN receiver, the RUBICON shares the knowledge acquired through the robot and can recognize in which room the user is located by looking at the signal strength information captured by the user's bracelet.
NAVIGATION - FAILED CAMERA	The user fits new curtains in the kitchen, so that the camera-based navigation mounted on the ceilings of that room needs artificial lighting to work properly. The RUBICON learns that it needs to switch on the light in the kitchen whenever the robots have to navigate there.
NAVIGATION-LOCALIZATION WATERING	Sometime before the camera-based localization system is repaired/replaced, the user waters the plants in the corner of the same room. After awhile, the RUBICON needs to send a robot to the kitchen. The RUBICON learns to send the wheeled robot, as it does not want to disturb the plant and would be very difficult to localize the Nao without switching on the light.

Table II - Case Studies in Hospital Transport Scenario

Name	DESCRIPTION
LASER/RSSI NAVIGATION	The robots employs a navigation system able to exploit both laser and WSAN RSSI information for localization purposes. In the first runs the positions of individual motes in are unknown and the system relies solely on odometry and laser to localize itself with success rates that vary depending on the areas. In areas with many features (corners, pillars, etc.) the system works well and in other areas it may require manual interventions to reset the position of the robot. After a number of runs, an RSSI to XY mapping will have been established by learning the expected RSSI information given a specific belief in the XY position of the robot. The system will thus have partitioned (implicitly) the areas of operations into the categories (a) areas in which RSSI and laser based localization is needed. (b) areas in which laser based localization is sufficient, but RSSI could also be used e.g. in case of failure of the laser and (c) areas in which RSSI could not be used, e.g. due to unreliable RSSI readings. All robots will be able to share the RSSI mapping. As a result, the whole

	system will be able to carry out its operations more reliably, efficiently, and also require minimum human intervention.
ANOMALOUS RSSI INFORMATION	After some time of deployment the transport robot have learned to associate XY positions to RSSI information and have learned the expected deviations in RSSI information. One day when the robot is traversing a long straight tunnel under the hospital, relying on RSSI information for localization, it encounters abnormal RSSI values due to some temporary construction changes. By noticing this abnormal situation a warning is triggered and alternative route to the target is planned.
CONTEXT DEPENDENT ROUTE PLANNING	Initially, the robots has no preference when it chooses traversable paths in the hospital. However, after a number of runs, the RUBICON ecology notices that the robots take longer time and result in more error situations when traversing one of the tunnels between 12:00 - 13:00. This is caused by a large number of employees using this tunnel to get to the hospital lunch restaurant. The robot learns to select an alternative route during these hours. This results in an adaptive and context dependent route planning that allows all RUBICON robots to avoid these problematic areas.
PRE-EMPTIVE DEPLOYMENT OF ROBOT(S)	When the robots are not being used for a transport task the RUBICON ecology has a number of "maintenance goals", such as maintaining a steady battery level on the robot, but apart from that pose no other constraints in their activities or positions. After performing a number of runs, RUBICON learns a pattern of context (time) dependent incoming requests for transport tasks. Additionally, it learns the average times to execute these transport tasks given the different starting positions of the robots. By noticing that more of the lunch transport tasks are satisfied when robots happen to start close to the cafeteria during weekdays 12am - 1pm the system will learn to prefer to park the robots close to these areas around these hours. In the long term this is expected to lead to an increase in the general effectiveness of the system as the different practical case studies of the robots are learned.
ANOMALOUS CARGO	By associating the general navigation parameters (speed, acceleration) with the given transport tasks that the robot is executing RUBICON learns to recognize what is a normal execution of a task and what constituted an abnormal situation. After a number of runs, RUBICON have in this way learned that the robots typically navigate fast and efficiently when they navigate towards the laundry room in the hospital but that their acceleration and battery consumption is higher on the navigation tasks away from the laundry room (due to their load). During one such execution, the robot is navigating to the laundry room and up again - expecting to be filled with heavy textiles - but accidentally it carries instead an empty trolley back from the laundry. This triggers an abnormal situation, and a manual user intervention is requested

Table III - Case Studies for both Scenarios

Name	DESCRIPTION
RESOURCE ADDITION	The RUBICON system is installed in a new environment. After some time, its user(s) complains about RUBICON's quality of service, for instance, the RUBICON fails to help out in the kitchen, the vacuum cleaner keeps being activated at inappropriate times, or the transport robot fails to find alternative routes when it encounters crowded areas. The system is updated by adding a number of extra sensors, e.g. microphones and PIR sensors. After some time, RUBICON improves its quality of service.
RESOURCE ADDITION MOBILITY	The RUBICON system is installed in a new environment. However, the new user decides to try the system without mobile components (robots and/or pan & tilt microphones). After some time, the user(s) complains about RUBICON's quality of service. The user agrees on the installation of pan & tilt microphones and/or mobile robots. After some time, RUBICON learns to operate the new resources to investigate and confirm its understanding of each situation and improve the way it conducts its operations. If the user decides to remove the new resources, they will be favourably surprised, as the RUBICON keeps doing a good job.

2.2 Provided Interface Requirements

This section briefly reports the requirements associated with the functionalities/data/control that the Control Layer must provide to interface with the other layers of the RUBICON architecture. These requirements are extensively discussed in the deliverables focused onf the layers requesting them. However, in this section the same requirements are reported and examined from the perspective of the Control Layer in order to ease the analysis of other requirements.

Table IV - Provided Interface Requirements

NAME /ORIGIN	DESCRIPTION
R4->3.GOAL Requested from Cognitive Layer (D4.1)	The Control Layer will carry out high-level goals set by the Cognitive Layer. Goals will describe the status of the world desired by the Cognitive Layer, such as the air aspirator should be on or off (case study COOKING); the light should on or off (case studies EATING, LIGHTS, NAVIGATION FAILED CAMERA...), the blind should be open or closed (case study PLANT WATERING...); the robotic vacuum cleaner should clean the living room (case study CLEANING); the robot should go to the bathroom (case study WATER TAP) or transport its load to station *X* (case study PRE-EMPTIVE DEPLOYMENT OF ROBOTS). Goals will request the achievement of services in a timely and pro-active manner (e.g. demanding to switch on appliances and lights in the case studies COOKING, EATING). Goals should also be used to drive the Control Layer to test hypothesis,

	investigate situations and gather knowledge to assist the operations of the RUBICON system and to support learning. For instance, robot mobility should be exploited to enhance the RUBICON's observation of occurring events (as in most of the case studies and emphasised by the case study KITCHEN WALL), and also to build evidence of the relationship between multiple, and(possibly different sensor streams (combining RSSI and location information, in case studies RSSI-BASED ROOM LOCALIZATION and LASER/RSSI NAVIGATION, and the output of different microphones, in case studies COOKING, EATING...) The Cognitive Layer may associate priorities and temporal constraints to each goal, respectively, to communicate urgency (e.g. case study FIRE ALARM), and to specify deadlines (e.g. transport arrival time in the in-hospital transport scenario).
R4->3.STATUS Requested by Cognitive Layer (D4.1)	The Cognitive Layer requires the Control Layer for status feedbacks reporting the status of the progress of any assigned goals. This is due to the fact that a control action may take substantial time (e.g. case studies WATER TAP, CLEANING, ALARM), and interim feedback instances may be needed to enable the Cognitive Layer to re-evaluate assigned goals, and possibly update or change them.
R4->3.EXPLORATIONS	The Cognitive Layer is curious about new events and sets EXPLORATION goals for the Control layer for the purpose of developing continuous self-adaptation of the RUBICON ecology. For this reason, the Control Layer must gather field/environmental information (in terms of location information, landmarks/way-points being reached/crossed by the robots (e.g. rooms) to the Cognitive Layer.
R2->3,4.WIRING Requested by Learning Layer (D2.1).	The Learning Layer requires the definition of new learning tasks in terms of data sources to be used as inputs and as teaching signal to train the network to compute the output associated to the task. For instance, in order to learn to predict the switch in the LIGHTS case study, the Learning Layer must be informed that there is a relationship between the readings from nearby sensors (e.g. passive infrared sensor/ RSSI data) to be used as inputs to detect the approaching user, and the status of the light switch, to be used as teaching signal by the learning task. Similarly, in order to learn to recognize that the user is watering the plant in the PLANT WATERING case study, the Learning Layer must be informed that there is a relationship between the use of the watering can, the light measured in the room.
R2->3,4.TRAININGDATA Requested by Learning Layer (D2.1).	The Learning Layer requires that the component requesting the creation of a new learning task provides a number of training samples associating input data to desired outputs, that are coherent with the wiring specified for the task.
R2->3,4.CONTROL Requested by Learning Layer (D2.1).	The Learning Layer requires to be instructed about the creation of a new learning task and the docking or undocking of new devices to the RUBICON ecology by appropriate control messages. This requirement is illustrated by the case study RESOURCE ADDITION, where the Learning Layer needs to be notified about the presence of novel sensors, whose transducers might be used as inputs for a novel

	computational learning task, and whose computational resources may be used to increase the size of the Learning Network.

2.3 Non-Functional and Common Requirements

Here we list non-functional and common requirements, such as those dictated by openness, fault tolerance, distribution considerations and any requirement involving more than one or possibly all the layers in the architecture.

Table V - Non-Functional Requirements

NAME	DESCRIPTION
R3.RELIABILITY	It is the duty of the Control Layer to generate functionally correct behaviour to make the ecology achieve the goals set by the Cognitive Layer within the requested time requirements. The Control Layer will provide mechanisms to react to and counteract unexpected and hostile circumstances (e.g. closed doors, obstacles, obstructions in all the case studies involving robot mobility). This requirement is demanded by the importance of many of the case studies, such as FIRE ALARM, where the Control Layer must support the fast response of the RUBICON Ecology, and in all the case studies involving robot mobility, where the Control Layer must avoid posing any danger to the user(s).
R3.HETEROGENEITY	The Control Layer will support heterogeneous robotic ecologies with varying computational constraints, as target environments will contain devices such as computers with large processing and bandwidth capacities (e.g. server(s) in the in-hospital transport scenario), as well as much simpler devices such as mini-PCs used in the AAL scenario, and also micro-controller-based actuators and sensor nodes, and even devices with no (customizable) computational capability at all, such as Radio Frequency Identifications (RFIDs).
R3.OPENESS	The Control Layer will support open robotic ecologies, where components (robots, appliances, sensors, actuators...) join and leave the system, for instance, as result of system maintenance, component failure and mobility. The requirement is also illustrated in case studies RESOURCE ADDITION and RESOURCE ADDITION MOBILITY.
R3.ROBUSTNESS	The Control Layer will provide mechanisms to minimize the effect and show a graceful degradation of the performances of the Robotic ecology in the presence of events such as component failures (e.g. case study NAVIGATION - FAILED CAMERA) and component unavailability (e.g. case study ALARM WHEN CLEANING) due to the open nature of the RUBICON Ecology.
R3.SCALABILITY	The Control System will provide mechanisms to promote the scalability of the system, in terms of number of goals that can be achieved at the same time and number of resources that can be co-ordinated to work toward their

	achievement. The performance of the Control Layer should degrade gracefully as workload of the system increases. This requirement is illustrated by robotic ecology employed in an AAL scenario, which should account for all the case studies listed in Section 2 (and more) at any given time and be ready to deliver all the associated services, and by the robotic ecology operating in the in-hospital transport scenario, which should be able to account for multiple robots and multiple transport requests and transport tasks active at the same time.
R3.DISTRIBUTION	At least some parts of the Control Layer will be distributed across the system (e.g. in close proximity of sensors and actuators and the distributed sections of the Learning Layer) in order to minimize communication and thus reduce network bandwidth usage, latency, and energy consumption. This requirement is especially illustrated in the case study LIGHTS, where it would be highly inefficient and energy consuming if the Control Layer had to monitor RSSI data (in the case the user wears a WSN node, as in RSSI-BASED ROOM LOCALIZATION) or data gathered from presence sensor in order to infer where the user is going before switching the lights in the apartment.
R3.ADAPTATION	One of key requirement for all the layers of the RUBICON architecture is to support system adaptation in order to increase the performance, the reliability, and the robustness of the system over time. This requirement is illustrated in most of the case studies, especially in the RESOURCE ADDITION and in the RESOURCE ADDITION MOBILITY case study. In general, it is expected that the RUBICON ecology will not be able to recognize with sufficient precision the activity of the user in the AAL scenario, to exploit RSSI data for robot localization, or select the best route in the in-hospital transport scenario as soon as it is installed in these environments. However, system performances will improve as the system will accumulate and share observation and experiences over time.
R3.KNOWLEDGE SHARING	One of key requirement for all the layers of the RUBICON architecture is to support knowledge sharing between all of its participants. This requirement is illustrated by the use case RSSI-BASED ROOM LOCALIZATION, where the laser-based localization is used to build evidence to support RSSI-base localization, and where the ability to track the position of the robot is applied to the tracking of the user. It is also illustrated by the case studies RESOURCE ADDITION and RESOURCE ADDITION MOBILITY, where newly added resources (robots, sensors) are expected to leverage knowledge learnt by the system before their introduction.

2.4 Other Functional Requirements

Here we describe the requirements set for the Control Layer that are not directly imposed by the interface with the other layers, as justified by the DoW, provided requirements and scenarios.

Table VI - Other Functional Requirements

NAME	DESCRIPTION	
R3.EXECUTION & CONFIGURATION	In order to achieve the goals set by the Cognitive Layer [R4->3.GOAL], and support the non-functional requirements outlined in Section 2.2, the Control Layer must co-ordinate the execution of actions and configuration strategies involving multiple robots and devices. In order to support R3.RELIABILITY, R3. HETEROGENEITY, R3.ROBUSTNESS, and R3.ADAPTATION, the Control Layer should find and consider different action and configuration options to achieve the same results. For instance, in the WATER TAP case study the Control Layer may find the following 2 options to monitor the bathroom: "*Option A = Move the Pioneer robot out of the kitchen using the localization information sent by the ceiling camera, cross the corridor, open door, and steps into the bathroom*, *Option B = Move the Nao robot out of the living room and use RSSI data to locate the bathroom...*". This will allow the proposal of alternative options to solve temporary impasses of the system, and also to explore different action and configuration strategies in order to ease system's learning.	
	DOMAIN KNOWLEDGE	In order to support R3.EXECUTION & CONFIGURATION, R4. HETEROGENEITY, R3.KNOWLEDGE SHARING and R3.OPENNESS, the Control Layer will deal with abstract knowledge of the types of resources, users, devices, robots and their capabilities (e.g. a Pioneer robot can move and localize itself, a ceiling camera can be used to estimate the position of a robot, a microphone can be used to detect sound events...).
	RESOURCES	In order to support R3.EXECUTION & CONFIGURATION, R3.KNOWLDGE SHARING and R3.OPENNESS, the Control Layer must have an up-to-date picture of the resources currently available in the system (software components, robots, actuators and sensors on both robots, WSN nodes and other devices).
	STATE	In order to support R3. EXECUTION & CONFIGURATION and R3.OPENNESS, the Control Layer must have an up-to-date picture of the state of the ecology (including the location of its participants), and of the state of the environment.

		This requirement is illustrated in the case studies FIRE ALARM, WATER TAP and CONTEXT DEPENDENT ROUTE PLANNING, in which the Control Layer needs to be informed of robot and user location, or if the door of the bathroom is open or closed, in order to drive the robot to its intended target location.
	SKILLS	In order to support R3.EXECUTION & CONFIGURATION and carry out its actions and configuration strategies, the Control Layer will leverage a set of pre-defined skills, i.e. available implementations of the capabilities of the participants of the ecology (as per R3.EXECUTION.MODEL). In all the case studies the robots will use pre-defined implementations of path-planning, safe navigation behaviours, localization and sound-recognition.
	REACTIVITY	In order to support R3.EXECUTION and R3.RELIABILITY, the Control Layer must be able to react to arising circumstances and events, such as obstacles along the path of the robot, component failure, and also opportunities arising from the intervention of the user or other robots. This requirement is illustrated by many of the case studies, including FIRE ALARM, BROKEN MICROPHONE, ALARM WHEN CLEANING, ANOMALOUS RSSI DATA and ANOMALOUS CARGO.
	PROACTIVITY	In order to support R3.EXECUTION, the Control Layer will be able to control and configure the RUBICON ecology in a pro-active manner. It is expected that the principal sources of the proactivity of the RUBICON Ecology will be given by the goals set by the Cognitive Layer (see R4->3.GOALS), and by the prediction provided by the Learning Layer (see R3,4->2.EVENT, R3,4->2.WEIGHT, R3,4->2.SENSORFUSE). However, the Control Layer will provide look-ahead and scheduling capabilities to directly support system proactivity. This requirement is especially illustrated by the case study PRE-EMPTIVE DEPLOYMENT OF ROBOT where the Control Layer will schedule and coordinate the robots in order to maximize the global throughput of the transport service and minimize the average waiting time.
	MONITORING	In order to support R3.EXECUTION.REACTIVITY, R4->3.STATUS, R4->3.EXPLORATIONS, R2->3,4.TRAININGDATA, the Control Layer needs to be able to monitor its own execution and assess its own performance in carrying out its plans.

		This must happen: • at runtime, in order to give interim status feedback to the Cognitive Layer and give it the opportunity to re-assess the goals of the ecology. • once the goals are achieved, to inform the Learning Layer of the quality of the outputs that were used to inform the choice of any of the options explored by the Control Layer (see R3->2.WEIGHT)

2.5 Requested Interface Requirements

Table VII - Requested Interface Requirements

NAME	DESCRIPTION
R3->1.SENSING From R3.STATE, R3.RELIABILITY, R3.EXECUTION & CONFIGURATION Requested to Comm. Layer (D1.1).	In order to build and maintain an up-to-date picture of the state of the robotic ecology and its environment (R3.STATE), and to enable collaboration between members of the robotic ecology (e.g. communication of localization data from the ceiling camera to the robot, in the AAL scenario, see R3.EXECUTION & CONFIGURATION), the Control Layer must be able to receive data and periodic status updates from every sensor and . In order to support R3.RELIABILITY, the Control Layer should also be able to specify the desired update rate and be informed of the maximum latency to be expected by the resulting updates. The Control Layer may tolerate the loss of some of these updates but all data must be timestamped in order to be able to ignore old updates.
R3->1.ACTUATION From R3.EXECUTION & CONFIGURATION, R3.RELIABILITY Requested to Comm. Layer (D1.1).	The Control Layer must be able to send control instructions (e.g. new set points, new output values) to every actuator . For this type of transmission, the Control Layer does not require the ability to communicate periodic updates of control instructions. However, in order to support R3.RELIABILITY, transmission of control instructions should be reliable (acknowledged). In addition, the Control Layer needs to be informed of the maximum expected latency.
R3->1.DATA SHARING From R3.KNOWLEDGE SHARING, R3.DISTRIBUTION, R3.EXECUTION & CONFIGURATION,	In order to support R3.KNOWLEDGE SHARING, R3.DISTRIBUTION and R3.EXECUTION & CONFIGURATION, the Control Layer must be able to (asynchronously) share sensor data, actuator status and other information among distributed nodes (i.e. multiple robots, WSN nodes and other devices).

R3->1.MESSAGES From R2->3,4.CONTROL R3.DISTRIBUTION, R3.EXECUTION & CONFIGURATION,	In order to support R2->3,4.CONTROL, R3.DISTRIBUTION, and R3.EXECUTION & CONFIGURATION and co-ordinate its operation across distributed nodes, the Control Layer must be able to send reliable and synchronous control messages to all the nodes .
R3->1.DISCOVERY & TOPOLOGY From R3.OPENNESS, R3.RESOURCES, R3.STATE	In order to support R3.OPENNESS, R3.RESOURCES and R3.STATE, the Control Layer needs an update picture of all the components available in the system, including all the WSN nodes currently active. Every component should have a unique ID and the Control Layer should be informed whenever any robotic device or WSN nodes join (as they become operative and connect to the network), or leave the system (as they get disconnected, breaks, they battery get depleted or simply move out of network range).
R3->2.EVENT From R3.RELIABILITY, R3.ADAPTATION, R3.STATE, R3.REACTIVITY, R3.PROACTIVITY To Learning Layer (D2.1)	In order to support R3.RELIABILITY, R3.ADAPTATION, R3.STATE R3.REACTIVITY and R3.PROACTIVITY, the Control Layer needs to be informed of events and high-level assessments of the status of the system, the environment, and the user. For this reason, the Learning Layer is requested to provide reliable prediction of the classification of (possibly) multiple events out of a predefined set of candidate events that might be occurring within the RUBICON Ecology. The Control Layer will use the information associated to these events to devise situation-dependent, pro-active action and configuration strategies and also to account for contingencies and opportunities arising during the execution of these strategies. This requirement is illustrated by the COOKING and EATING case studies, where the Learning Layer is trained to predict the event of the user preparing food based on the input from several sensors in the kitchen.
R3->2.SENSORFUSE From R3.RELIABILITY, R3.ADAPTATION, R3.STATE, R3.REACTIVITY, R3.PROACTIVITY To Learning (D2.1).	In order to support R3.RELIABILITY, R3.ADAPTATION, R3.STATE R3.REACTIVITY and R3.PROACTIVITY, the Control Layer needs reliable and accurate perception data. This information should be obtained with the aid of multiple sensor and data processing sources and should improve the quality of that obtained from raw sensor data (R3.SENSING), i.e. in terms of accuracy and signal-to-noise rate. For this reason, the Learning Layer is requested to process input sensory data streams and to fuse them into a prediction of a (possibly) different Service measure. This requirement is illustrated in the RSSI-BASED ROBOT's LOCALIZATION case studies, where the Learning Layer is trained to predict room occupation

	for the robot based on signal strength information (RSSI) from its radio devices; in the KITCHEN WALL case study, where the robot's sensors (e.g. microphones) will be used to recognize sound events and consequently improve the sound recognition performed using the microphones already located in the kitchen.
R3->2.WEIGHT From R3.RELIABILITY, R3.ADAPTATION, R3.STATE, R3.REACTIVITY, R3.PROACTIVITY To Learning (D2.1).	In order to support R3.RELIABILITY and R3.ADAPTATION, and R3.PROACTIVITY, the Control Layer needs help in prioritizing action and configuration options to find the most suitable action or configuration strategy, i.e. the one with more probability to succeed and to give good results (e.g. in terms of time efficiency, resource usage, user's satisfaction, etc...). For this reason, the Learning is requested to provide a set of weights that can be used by the Control Layer to predict the suitability of its strategies to pursue its goals. This requirement is illustrated in the NAVIGATION CARPET case study, where the Learning Layer is trained to suggest which robot to send to the bathroom.
R3->2.REFINEMENT From R3.ADAPTATION To Learning (D2.1).	In order to support R3.ADAPTATION, the Learning Layer is requested to support the refinement of its predictions (of existing LN outputs (associated to R3->2,4.EVENT, R4->2,4.WEIGHT and R3->2.SENSORFUSE) throughout appropriate teaching signals (see R2->3,4.TRAINING DATA). This requirement is illustrated and discussed in more details in R3.ADAPTATION.
R3->2.INCREMENTAL From R3.ADAPTATION To Learning (D2.1).	In order to support R3.ADAPTATION, the Learning Layer, the Learning Layer is request to support the incremental definition of new outputs (associated to R3->2,4.EVENT, R4->2,4.WEIGHT and R3->2.SENSORFUSE). Such an incremental addition must be performed within an already deployed and currently active RUBICON ecology. This requirement is illustrated by the PLANT WATERING case-study, the Learning Layer is instructed to create a new output (of the type in R4->2,4.WEIGHT) that is trained to predict a weight for the control action of closing the blinds. Similarly, a new output is created to predict a weight for the lights-off action. The two newly created weights are used by the Control Layer to help determine which action, closing the blinds or switching-off the lights, is more likely to achieve the RUBICON goal of making the room dark after the user has watered the plants.

3. Background Tools and Techniques

Within the RUBICON project we will rely on background technologies coming from ORU and from UCD within the Control Layer (WP). The first of these (ORU) consists of the PEIS Ecology middleware and associated action & configuration planners and constraint reasoning based controllers. The second of these (UCD) consists of agent system technologies that will be merged with the former in order to gain scalability and robustness through decentralization of the tasks.

We will in this section only briefly describe the most important aspects of these technologies required for understanding of the high level design presented in Chapter 4. For the interested reader we refer to the websites of the PEIS middleware[1], of Agent Factory[2] and of the publications by the contributing authors for more complete descriptions.

3.1 PEIS

We describe here the main technologies coming from the Ecologies of Physically Embedded Intelligent Systems project (PEIS Ecologies) from ORU. Some of these descriptions are overlapping with the descriptions presented in Deliverable D1.1, but here targeted mainly towards the control tasks.

3.1.1 Introduction

The PEIS kernel and related middleware tools are a suite of software previously developed as part of the *Ecologies of Physically Embedded Intelligent Systems* project in order to enable communication and collaboration between heterogeneous robotic devices. This kernel is a software library written in pure C and with as few library and RAM/processing dependencies as possible in order to fit on a wide range of devices. The original purpose of this library was to enable software programs running on PC (Linux, MacOS, Windows), PC/104 (Linux/RTAI), Gumstix (uCLinux) to participate as PEIS components in the PEIS Ecology network. This network is comprised of a heterogeneous set of mobile robots and networked sensors and actuators. The middleware is used to enable communication and collaboration between these devices that are in many aspects non-overlapping (and orthogonal) to hardware centric robotic middleware such as Player/Stage and ROS.

Within the frame of RUBICON this middleware will be used both as an integral part of communication between robotic devices in WP1 and for enabling higher level collaboration aspects such as abstract subscriptions and dynamic re-configurability.

The previously existing implementations of the PEIS middleware satisfies the requirements R3.HETEROGENEITY, R3.OPENESS, R3.SCALABILITY and R3.DISTRIBUTION. Furthermore it directly supports R3.KNOWLEDGE SHARING and many of the R3.EXECUTION & CONFIGURATION sub-requirements - as outlined in the descriptions below.

In the remainder of this section we will describe the control and collaboration aspects of the PEIS middleware as applicable to the RUBICON project. For further details and for the communication aspects, we refer the reader instead to Deliverable D1.1.

[1] http://www.aass.oru.se/~PEIS, http://www.aass.oru.se/~{mbl, asaffio, fpa}
[2] http://www.agentfactory.com

3.1.2 High-level PEIS Kernel Design

The most important design requirements of the PEIS-kernel have been to provide a decentralized mechanism for collaboration between separate processes running on separate devices that allows for automatic discovery, high-level communication and collaboration through subscription based connections and dynamic self-configuration. These and any additional services should all allow any devices to communicate/collaborate with any other devices as long as there exists any, possibly indirect path of communication between the devices.

From the point of view of an application programmer, the PEIS kernel and middleware provides a number of core services available to programs linked against the PEIS-kernel as well as a few dedicated middleware services implemented as programs that can be run on each participating PC or robot.

The basic services provided by the PEIS middleware is communication links between any PEIS-components (processes linked against the PEIS-kernel library) running on the same machine or machines reachable through any direct or indirect link – with a selection of different primitive link types including TCP/IP and Bluetooth – through the use of a dynamically established P2P network.

On top of these communication services a set of higher level services, most notably a shared tuplespace service that allows for high level collaboration between different devices. In the remainder of this section we will primarily focus on these high-level services and refer the reader to deliverable D1.1 for details on the internal communication mechanisms.

3.1.3 Overview of the Tuplespace

From the point of view of application programmers, a *tuple* is a *key-value* pair that can associate any piece of data to a logical key. In the PEIS-ecology we enable these tuples to be shared between any devices within communication range and enforce a few additional constraints to the format of the tuples and to associated meta-information such as timestamps and MIME types of the data.

The tuplespace service of the PEIS-kernel is responsible for all storage, publishing and retrieval of *tuples* in the distributed tuplespace – effectively creating a distributed database as a blackboard communication and collaboration model. By performing associative wildcard searches it allows for efficient collaboration between any pair of components in the ecology. By introducing the concepts of meta-tuples allowing for indirect access to data it enables simple and efficient dynamic reconfiguration of inputs and outputs.

Although needed for high-level collaboration the kernel does not force any specific format of the shared data, but define only loose standards to the used formats and semantics. For the specific applications developed in an application domain the semantics and formats of the shared data will follow a lightweight formal format as needed to be understood by the other components.

From the perspective of the tuplespace, keys consists of three parts: (name, owner, data) where *name* is a string key for the tuple, *owner* is the address of a PEIS responsible for this tuple (see below) and data is the value of the tuple. The tuples are indexed by *name* and *owner* meaning that tuples with the same name but different owners are allowed to coexist while there can (ideally) only be one instance at a time of tuples with the same *name*, *owner* (but different data).

For further details regarding the semantics of the distributed aspects of the tuplespace we refer the reader to Section 3.1 in deliverable D1.1.

Input streams of data are often accessed by explicitly querying and reading the latest value of a tuple in the tuplespace within the main loop of a robotic program. Although this method is convenient a more efficient method with fewer drawbacks such as a risk of missing some values or reading the same value twice is to use callback functions.

By registering a callback function with an *abstract tuple* as a prototype for the kind of tuples of interest a given function will be guaranteed to be invoked with the matching tuples as they arrive into the local cache of the PEIS kernel.

By registering callback functions triggered by any failure signals of the components launched on the same host, the RUBICON's Control Layer will be able to detect failures and respond correspondingly by triggering a re-planning or signalling failures to higher levels.

3.1.4 Meta Tuples

Although presented also in Deliverable D1.1 we remind here the reader of the notion of meta tuples [70] since these are expected to play an important role in the *configuration* of WP3.

A meta tuple is a tuple which gives the owner and name of other tuples. Thus meta tuples provide a mechanism for *indirect reference* which allows components to be dynamically reconfigured by rewriting the references during execution.

In the most simple scenario for executing a collaboration between components, producers create data in their own tuple space and consumers establish subscriptions to these tuples to access the data to be used. However, since consuming components cannot know in advance from where to read the data to be used – a *configuration* must be used to establish which components are connected. Meta tuples are a mechanism for allowing this in a general way. By using these as inputs it is possible for consumers to read hard coded meta tuples from their own tuplespace. This corresponds to meta tuples acting as named input ports in other middleware.

To configure such a consumer, a configuration writes the id and key of tuples produced by any producer. The consumer will then automatically subscribe to and read the data from the producer. From the users point of view this makes programming configurable components very simple since *input* tuples then can be read with a simple API call and the kernel automatically handles setting up and removing new subscriptions as the configuration is changed.

Example using *pseudo-code*:

```
Producer 42:
  while 1:
    setTuple "temperature" <- sensorReading()
Consumer 22:
   subscribeIndirectTuple(peisid(),"heat")
   while 1:
     T = findIndirectTuple("heat")
     if(T)
       do-something(T->data)
```

```
Configurator:
   setTuple "22.heat" <- "(META 42 temperature)"
```

Apart from using meta tuples to create indirect input values, it is also possible to use it to create indirect write values. By using this functionality is it possible to have a component *A* which is an alarm clock connect its output tuple to component *U* which is a user. Typically, the alarm clock does not create tuples, but when it does the output would be written to a specific tuple in *U* 's tuple space. In this way it is possible for *U* to be receiving alarms from many components. For events and signals it is more intuitive to allow components to push values to others, and not just pulling in tuples.

This concept supports the R3.OPENESS and R3.PROACTIVITY requirements in that components that can be re-configured to use new sources of data whenever the topology of the ecology changes. Furthermore in WP3 these meta tuples will be used to reroute functionality flow between the robotic components launched by the service layer BDI functionality. By using input and output meta tuples in all the component launched to perform the sensing and actuation tasks this BDI can route functionality flows directly between components onboard the same host (agent) and information flow that is to pass to remote agents.

3.1.5 PEIS-init

PEIS-init [71] is a middleware component with some access to internal functionality of the PEIS-kernel. It's main purpose is to be a central location on each host to store semantic information about components (programs) that can be run on that host, and to starts or stop and monitor the execution of these programs.

- It should automatically be started on all machines in the PEIS ecology when they boot.
- It should be able to start and stop components on the same PEIS using special tuples showing the requested state for that component.
- Monitor the state of all started components and restart them if necessary. Contain tuples showing their standard input / output / errors etc.
- Displaying the contents of the tuples, with their standard input, output and errors.

The PEIS-init component relies on a set of .cmp files on the local machine to determine which component can run on it and what their semantic descriptions are. These descriptions are exported to the tuplespace to be used by configuration planners on any machine in the ecology.

For each possible component, PEIS-init subscribes to tuples to set the start, stop or restart state of the components. It forks and executes the corresponding software components if the components are requested to be run, monitors their inputs, outputs and execution states (restarting them if necessary) and stops the components when they are no longer needed.

When asked to keep the topology of the P2P network simpler, PEIS-init uses the leaf-mode option when starting other component and routes their network traffic through itself.

By default, PEIS-init is started automatically when a machine is booted up. When additional software components are compiled and installed they add their corresponding entries to the configuration directory of PEIS-init and they become available to the ecology.

This program will become an integral part of the RUBICON's Control Layer *that runs on each host. It will be used to start and stop services on the local machine (see Section 4).*

3.1.6 Action & Configuration Planning

Although classical AI planning such as STRIPS based operators can easily be extended to work for ecologies of collaborative devices these planning methods suffer from a number of challenges that make them less than ideal for the implementation of the RUBICON ecology.

The first of these challenges is due to the demands on robustness combined with a demand of combinatorial generality – where the possible combinations of devices should be able to provide functionalities to assist other devices is expected to grow superlinearly as more devices are added to the ecology.

Computing which actions are to be performed by individual devices are traditionally delegated to an action planner that reasons about the possible outcomes of different actions on a given model of the environment. As the environments become increasingly complex, unstructured and with increasing demands of methods for accurately handling errors in perception or actuation these planning models tend to increase in complexity and to become intractable.

We call the set of devices that are actively exchanging data in a collaborative fashion at any given time the *configuration* [71] of the ecology. The task of computing the configuration to be used at any given time in order to accomplish the actions generated by an action planner can also be modelled explicitly as a search problem and solved either in a dedicated configuration planner or as an integral step of the action planners. For this purpose such configuration planners typically rely on introspection and semantic descriptions of the available components in order to create a *domain description* that includes all the available devices and the actions and data-exchange functionalities that they support. This is illustrated Figure 2 , below, where a configurator plans for a subset of the available devices to perform specific localization tasks in order to assist the robot Astrid to navigate and open a refrigerator door.

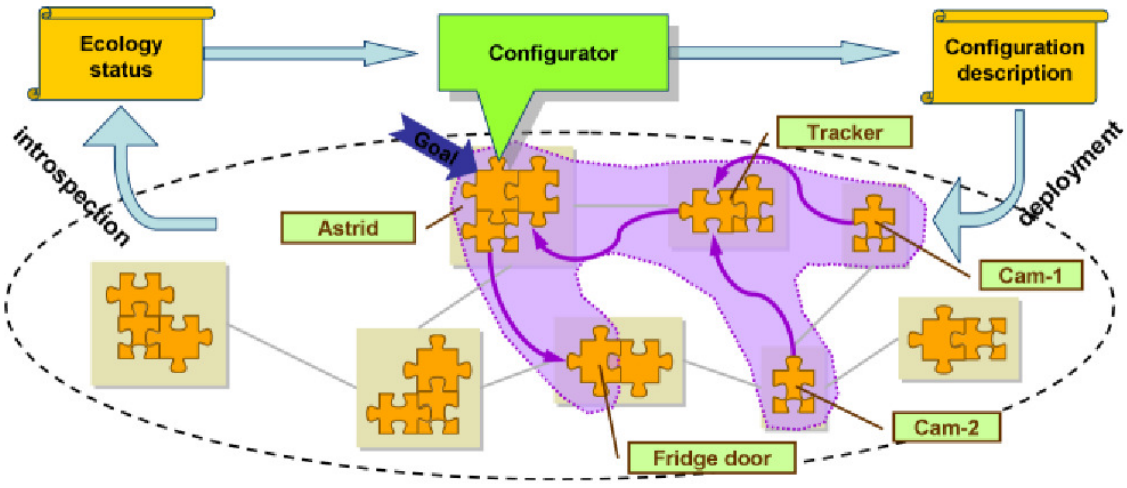

Figure 2. An example of a Configuration Plan in PEIS

Although some success [72] have been made in the literature for solving the joint action and configuration planning problems at a global level for heterogeneous robotic ecologies we propose within RUBICON to distribute certain aspects of these tasks over a number of agents with varying

responsibilities and functionalities. As we will see in Section 4, "High-Level Design", we expect this to lead to better scalability, robustness and fault tolerance.

3.1.7 Temporal Constraints Reasoning Approach

Timeline-based planning is relevant to the proposed RUBICON control infrastructure. Such approaches are particularly meaningful in our context as we are interested in generating plans that can be monitored during execution by the BDI agent. Timeline-based planners are suited for monitoring because they retain rich temporal information . This information can be used by the BDI agent to ascertain in a timely fashion which task within the plan is currently in execution, as well as the precise temporal placement of all future elements of the plan. In essence, timeline-based planners lend themselves to bridging the gap between task planning and execution monitoring. Timeline-based planners leverage constraint-based reasoning algorithms to propagate temporal, resource and causal relations. Also, most existing implementations of such planners are highly modular and extendible. Planners such as Europa [53] and OMPS [54] have been employed to control complex systems and processes such as space mission control for the US and European space agencies respectively. An example of how such planning frameworks can be used to bridge the gap between planning and execution is shown in [53] . More recently[52] , a similar approach has been extended to perform situation assessment and contextual plan generation and execution. This is particularly meaningful in RUBICON because it opens the possibility to specify sophisticated models for “expected” execution of plans. These models can be used to infer high-level diagnoses of faults in plan execution, which can in turn be used by the BDI agent for fault tolerance.

Several research groups have developed constraint-based planning starting from an application perspective. A temporal planning architecture deployed on the flight processor of a spacecraft in NASA’s Remote Agent Experiment [55] provided a first example of closed-loop planning and execution on a real system. This has been followed by an increasing number of architectures for reasoning in application settings where dynamic, real-world constraints demand reasoning algorithms that are flexible and efficient. Among these, space mission planning [56] [57] [58] , robot plan execution architectures [59] , planning for intelligent environments [52] and unmanned aerial vehicle control [60] . Interestingly, all these systems leverage temporal constraint reasoning algorithms [61] to solve the temporal aspect of the overall problem.

Given the previous successes of constraint-based planning methods in real-world contexts and in robot control frameworks, we expect these techniques to be most suitable for the implementation of the planning and monitoring part of the RUBICON's Control Layer.

3.2 Agent Systems

Agent and Multi Agent Systems (MASs) are regarded as a general-purpose paradigm to facilitate the co-ordination of complex systems built in terms of loosely-coupled, situated, autonomous and social components (the *agents*). In particular, the Belief Desire Intention (BDI) agent model provides a simple but extensible model of agency that explicitly addresses the production of rational and autonomous behaviour by agents with limited computational resources. The vast availability of BDI-based toolkits, such as OpenPRS, JACK, Jadex, Jason (see [1] for a review and references), and the suite of agent toolkits developed at NUID UCD, offers a suitable ground for tackling the many operative requirements of robotic ecologies while applying state-of-the-art software engineering techniques.

These methods and previously existing NUID UCD implementations of the Agent Factory Multiagent framework help satisfying the requirements R3.HETEROGENEITY, R3.OPENESS, R3.SCALABILITY, and R3.DISTRIBUTION. Furthermore they directly supports R3.KNOWLEDGE SHARING, R3.ROBUSTNESS, R3.RELIABILITY, and many of the R3.EXECUTION & CONFIGURATION sub-requirements - as outlined in the descriptions below.

In order to inform the design of a decentralized, scalable and robust control solution for the RUBICON ecology, this section provides a brief overview of the BDI agent model and of some of the implementations available to the consortium. It also examines related works to see how these systems have been integrated with planning and learning capabilities in order to satisfy R3.PROACTIVITY and R3.ADAPTATION.

3.2.1 The BDI Abstract Architecture and the Procedural Reasoning System (PRS)

Rao and Georgeff 's formalization of the BDI model [32] [30] [32] originated from the theory of human practical reasoning developed by the philosopher Micheal Bratman [3] . The model focuses upon three attitudes, respectively representing the information (*beliefs*), motivational (*goals* and *desires*), and deliberative (*intentions*) states. The intentions of an agent are subsets of its beliefs and desires, i.e., an agent acts towards some of the world states it desires to be true and believes to be possible.

The essential difference between practical reasoners and planning systems is that the first do not explore all the possibilities, evaluating all their available choices and the possible outcomes of the subsequent interaction with their environment, before selecting an action to perform. In contrast, action-selection is structured in the two successive phases of *deliberation* (generating options to pursue) *and means-end analysis* (deciding how to pursue these options). This is similar to the structural partiality of plans in hierarchical planning but with the fundamental distinction that in practical reasoners the expansion of partial plans is interleaved with their execution in order to cope with contingencies in dynamic or unpredictable environments.

For its similarity with the successive realization of implementations of Rao and Georgeff's abstract BDI interpreter in modern agent systems (including those developed at NUID UCD), it is worth illustrating Georgeff and Lansky's Procedural Reasoning System [19] .

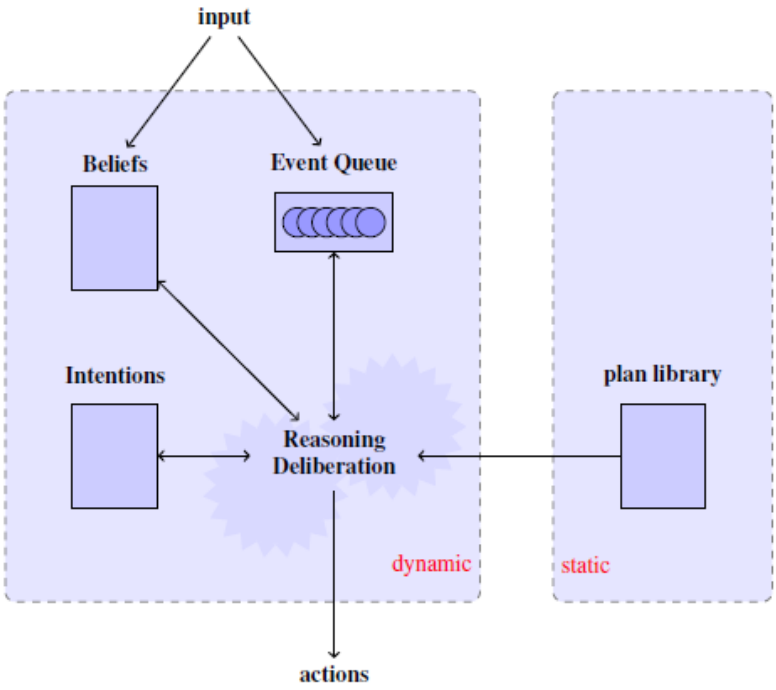

Figure 3. Architecture of the Procedural Reasoning System.

PRS implements a computationally tractable BDI model with the following simplifying assumptions:

- The system explicitly represents beliefs about the current state of the world as a ground set of literals with no disjunctions or implications (as in STRIPS [16]).
- The system represents the information about the means of achieving certain future world states and the options available to the agent as pre-compiled plans.

Each plan in the plan library can be described in the form e: $\Psi \leftarrow P$ where P is the body of the plan, e is an event that triggers the plan (the plan's post-conditions), Ψ is the context for which the plan can be applied (which corresponds to the preconditions of the plan). The body of each plan is a procedural description containing a particular sequence of actions and tests that may be performed to achieve the plan's post-condition. Plans may also post new goal events, leading to the characteristic goal-plan execution trees depicted in Figure 1. These structures can be seen as AND/OR trees: for a plan to succeed all the subgoals and actions of the plan must be successful (AND); for a subgoal to succeed one of the plans to achieve it must succeed (OR). When a plan step (an action or subgoal) fails for some reason, this causes the plan to fail, and an alternative applicable plan for its parent goal is tried. If there is no alternative applicable plan, the parent goal fails, cascading the failure and search for alternative plans one level up the goal-plan tree.

The PRS interpreter runs the main loop of PRS by keeping track of the state of the set of plans currently active in the system and updating its own internal state with the following algorithm:

1. Select an event
2. Update beliefs to account new information gathered from the environment (e.g. through sensing).

3. Determine a set of ”applicable plans” to respond to the new event.
4. Select one applicable plan and add it to the intention structure. This may be a new intention or it may expand the details of a current intention.

   Execute the next step of a selected intention. This may execute an action or generate a new event.

The fundamental observation behind the decoupling between goals and plans is that goals, as compared to plans, are more stable in any application domain and multiple plans can be used or attempted to achieve the same goals. This also allows examining the application domain in terms of what needs to be achieved, rather than the types of behaviour that will lead to achieving it.

Goal events in PRS are posted by using special temporal operators like *achieve*, *preserve*, and *maintain.* The achieve operator is used to request the achievement of a new goal. For instance, an agent in control of a thermostat may be instructed with a new set-point temperature by posting the goal *achieve(T>20)*. Achieve goals are dropped either via an explicit requests (e.g. by the user), or whenever they are achieved, e.g. when the perceived temperature reaches the set-point. The *preserve* and the *maintain* operators specify a homeostatic goal - one that must be re-achieved if it ever becomes unsatisfied. Contrary to goals declared with its passive counterpart *preserve*, a *maintain* goal is automatically re-posted every time the condition it represents become unsatisfied. In the same thermostat example, the thermostat instructed with the goal *maintain(T<20)* may trigger the activation of an air conditioning system every time the temperature pops above the set point.

The search for alternative applicable plans when a goal is first posted or when a previously attempted plan has failed enables these systems to robustly recover from many problems, particularly problems where something has changed in the environment, motivating a different selection of plan. Furthermore, in some situations there can be multiple plan options to achieve a given goal, but for a given state, only certain combinations of choices will lead to its successful or satisfactory achievement.

The final decision of which plan to activate is performed using meta-level procedures that operate upon meta-level descriptions of the other procedures in the system. Meta-level plans are important hooks used by the designer to implement application-specific strategies, for example, by considering application-specific attributes defined as properties of the procedures (i.e. priorities, preferences) or insuring mutual exclusion on critical resources.

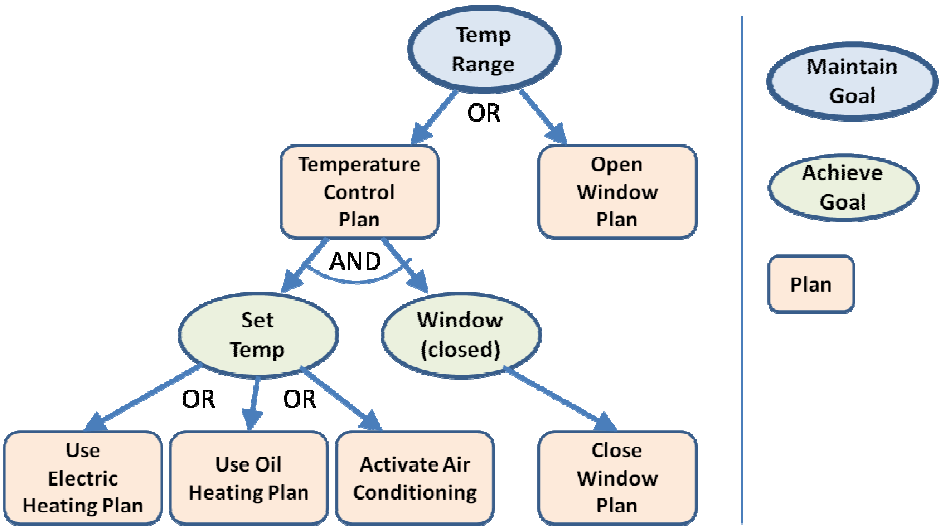

Figure 4. Example of BDI Goal-Plan Graph for a simple agent in charge of the temperature in a smart home

One of the difficulties with the BDI approach is that hierarchical goal decomposition, of the type advocated by Kinny *et al.* [23] and exemplified with the example in Figure 2, is most effective when there are no dependencies between the different sub-goals in the system. This is indeed the case assumed in Rao & Georgeff's theoretical BDI framework, which requires that goals are consistent for the theory to be valid. Unfortunately, this is also a very unrealistic expectation in most real-world applicative domains and especially in robotics where mobile robots are commonly subjected to conflicting goals (e.g. recharge batteries versus explore).

A PRS-like implementation of the BDI agent model will be used as part of the control solution in charge of executing and monitoring both action and configuration strategies in each participants to the RUBICON ecology (thus supporting R3.EXECUTION & CONFIGURATION and R3.MONITORING). Such a control solution will ease the efficient and robust application of these strategies to heterogeneous and computationally constrained devices (thus supporting R3.HETEROGENEITY, R3.ROBUSTENESS, R3.RELIABILITY). Adaptive plan selection will be afforded through the integration with learning and planning components (thus supporting R3.ADAPTATION and R3.PROACTIVITY).

3.2.2 Multiagent Coordination and Negotiation

A significant part of multiagent system (MAS) research focuses on improving the level of cooperation achieved between loosely coupled agents in order to maximally exploit their capabilities, minimize conflicts and resource contentions between them [63] .

To this end, MAS research is mostly focused on explicit coordination mechanisms based on symbol-based agent communication language (ACL) [66] messages that are used to influence the mental state of the agents that receive them. Typically, ACL-based communication does not consist of sending unrelated messages. Rather, dialogues take place, during which several related messages (such as *inform*, *request*, *accept* …) are exchanged between two or more conversation partners.

Coordination mechanisms in early MAS solutions were mostly formulated in terms of planning, seeking distributed solutions to the problems of centralised multiagent planning, namely their lack of scalability and robustness, and the difficulty of being applied in systems lacking global knowledge of their state.

Organizational approaches have been widely adopted as a MAS coordination mechanism. They are based upon the establishment of (temporary) hierarchical dependencies among some of the agents in the MAS and on the prescription of roles and communication protocols to every participant within the hierarchy. Economics has inspired the *Contract-Net Protocol (CNP)* [65] – one of the most renowned MAS coordination techniques for task and resource allocation - which distinguishes between agents asking to execute tasks and agents ready to handle them. The asking agents broadcast a call for proposals, and the helping agents submit their bids before the pending task is granted to the best bidder

MAS research has also produced many negotiation mechanisms for preventing or resolving conflicts between multiple agents. These mechanisms can be distinguished between competitive, which are particularly applicable to self-interested MASs where agents do not necessarily cooperate, and cooperative, which are suitable instead for cooperative applications, such as RUBICON. The majority of cooperative negotiation mechanisms are based upon the BDI formalism. These systems usually leverage upon the notions of social commitment [67] and related Joint commitment models [68] [69] , to define communication protocols for negotiations. In these approaches, the notion of agent commitment toward a goal is extended to include commitments toward other agents, intended as obligation or promises of the execution of certain actions or the adoption of certain objectives. Such social commitments can be used to define rounds of pre-commitment negotiations to allow agents to negotiate toward a joint commitment, before deciding to commit to a specific course of action.

The RUBICON Control Layer will be organized as a two tier, hierarchical multiagent system. This will enable WP3 to focus on the added value in RUBICON, i.e. adaptation, while providing a scalable control solution and an avenue to investigate the use of negotiation mechanisms for the efficient coordination of RUBICON's activities (thus supporting R3.HETEROGENEITY, R3.KNOWLEDGE SHARING, R3.OPENNESS, and R3.SCALABILITY).

3.2.3 Capabilities and Introspection

BDI systems such as Jack [5] and Jadex [29] have been extended with the ability to define and use descriptions of agent capabilities. Mainly, these systems are intended as suitable mechanisms with which agents in a MAS heterogeneous system can obtain and use information about other agents, for example before requesting services or during plan negotiation. With such mechanisms, an agent may then contact or try to influence agents with the required capability (to obtain help in some task), or alternatively may make decisions about its own actions based on the believed capabilities of other agents.

The general interpretation of an agent having a capability to achieve a given goal is that the agent has at least one plan that has as its trigger (post-condition) the goal. That is, the agent has at least one way that it knows how to achieve the goal in some situation. The capabilities management in both Jack and Jadex are implemented through the introduction of a *capability* construct used to package functionally related entities (beliefs, goals and plans), defining the interface to the related functions while filtering internal implementation details. A capability can then be simply specified by

a particular list of goal achievement events that the agent is designed to handle, abstracting from both additional sub-goal events and the plans addressing them.

The abstraction offered by capabilities in JACK, Jade and AF, is most useful in large and complex MAS as it eases the search for agents that are able to respond to specific events and possibly achieve specific goals. However, as Padgham & Lambrix note [28] , such systems preclude the automatic discovery of plans that are internal to a capability and are not exported by its interface - rarely to be represented as a static property of an agent. More likely, it may be possible for an agent to be able to achieve a goal only in determinate situations, e.g. when a needed resource is finally available, so that the precise assessment of an agent's own capabilities is necessarily part of the agent reasoning process [28]

Within the NUID UCD's AF framework, Ross [34] has investigated the latter view by developing the concept of *agent introspection* [34] , intended as the ability of an agent to analyze its own mental state and to perform backward planning to decide if it could attempt to achieve a goal if it was requested to do so.

The RUBICON's Control Layer will employ a mix of these techniques, by using plan-based introspection to find out the actual, context-dependent capabilities of any of the participants to the RUBICON ecology, and also to ease the distribution of control strategies among multiple agents (thus supporting R3.PROACTIVITY, R3.KNOWLEDGE SHARING, R3.DISTRIBUTION).

3.2.4 Agent-Planning Integration

The use of pre-defined plans in BDI architectures helps to avoid the computational complexity of planning in agents with limited resources by ensuring bounded time for means-ends reasoning. On the other hand, the resulting performance gain at the execution stage must be paid with the increasing cost of designing the plan library for each application and to handle every possible situation the agent may find itself in. Inevitably also, difficult domains lead to a combinatorial explosion of possible situations that need special sub-plans, which can soon bring out of hand the design process. One obvious way to add some automatism into the final application, and consequently alleviate the burden at the design stage, is to use external planners as run-time BDI procedures, as in many robotic architectures of hybrid design that use BDI reasoning in conjunction with specialized planners, for example for path-planning.

More general and integrated solutions, trying to incorporate action planners directly into BDI systems, are inherently more difficult in light of the relative inefficiency of general-purpose planners compared to domain-specific solutions. Such construction can follow two distinct directions, depending on whether the initiative resides in the BDI or the planning component.

An early example of the first approach is Ferguson's TouringMachine architecture [15] , where a BDI system is in control of the planner, which is called on-demand whenever the agent needs a course of action to achieve a given goal. The general issues raised by such integration schema are investigated by Walczak et. al. [41] using the Jadex BDI interpreter as system controller responsible for the upper part of the intentional structure. A preliminary step to this type of integration is dedicated to cancelling the differences between the plan representation in the BDI and the planner systems. Such a step is necessary because, although the body part of PRS's plan has some similarities with HTN [10] , sub-goaling in PRS's plans is also interleaved with some control programming structures (such as if-then else, while, etc), which prevent a direct integration with the HTN algorithms.

Sardina et al. [37] formally define how HTN planners can be integrated into a BDI architecture. Sardina shows that the HTN process of systematically substituting higher-level goal tasks until concrete actions are derived is analogous to the way in which a PRS-based interpreter pushes new plans onto an intention structure, replacing an achievement goal with an instantiated plan. Taking advantage of this almost direct correspondence, an HTN planner is used to add *lookahead* capabilities to an agent, allowing it to optimise plan selection and maximise an agent's chance of successfully achieving goals. By verifying beforehand the selection of plans for achieving subgoals, the agent minimises the chance of failure as a result of poor plan selection.

The process of converting information from a BDI system to a HTN planner is also addressed in [11] , with the JACK/JSHOP integration. The conversion between JACK programs into suitable JSHOP representations is a straightforward one-one mapping of JACK's goals and recipes (including each recipe's precondition and effect), into JSHOP's syntax (i.e. *methods*, and *precondition* and *tail* pairs). However, as with most HTN planners, JSHOP provides a totally ordered sequential plan, while BDI systems like JACK are intended to be able to pursue multiple goals in parallel, and use partial plans, i.e. plans specified in terms of sub-goals. The solution implemented in [11] is to modify the HTN planner to provide information on the planning process itself, i.e. for each operator or action in the plan, which recipe and goal instance were selected to lead to that action.

Further examples of such an approach include the the work of Ingrand and Despouys at LAAS/CNRS [13] , and the X-BDI (extended BDI) and X_2BDI (extended X-BDI) systems [14] . Ingrand and Despouys extend their own implementation of PRS, Propice, by adding explicit planning capabilities. The result, Propice-Plan, combines dynamic plan synthesis to complement existing operational procedures with anticipation planning to anticipate (through simulation) on the plans execution. Anticipation planning allows examining in advance the outcomes of various possible execution paths and consequently advising the execution layer or the best option to take when facing choices, and to forecast problems that may arise due to unforeseen situations.

In X-BDI the representation of agent internal mental states is mapped to the STRIPS planning notation. X-BDI is subsequently augmented with Graphplan. The planner in X_2BDI is integrated with the agent's goal deliberation and intention re-consideration process. Every desire in an X_2BDI agent is conditioned by a conjunction of literals called *Body*, which specifies the pre-conditions that must be satisfied in order for an agent to desire a property. Desires may be specified to be valid only in a specific moment, or whenever its pre-conditions are valid. Desires also have a priority value used in the formation of an order relation among desire sets. In order to reduce the frequency of intention reconsideration X_2BDI uses a set of reconsideration "triggers" generated when intentions are selected, and causes the agent to reconsider its course of action when activated. Specifically, intentions are re-considered only when the world changes in such a way as to threaten the plans an agent is executing or when an opportunity to satisfy more important goals is detected.

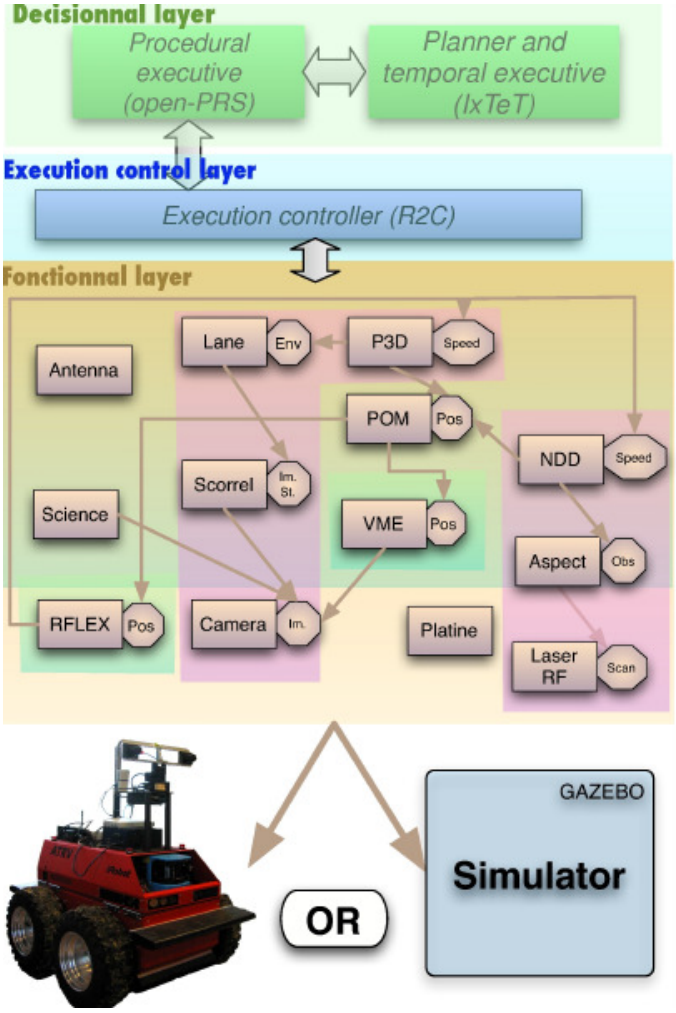

Figure 5. The IxTeT/LAAS robot control architecture(from [22])

More recently, the AgentSpeak agent interpreter has been extended with the ability to extend its plan library on-demand [51] . New plans are created by sequencing existing lower-level plans from the plan library, from which key information about their declarative preconditions and consequences is extracted. The resulting system, called AgentSpeak(PL), uses state-space planning to create new plans, and since state-space planners are inherently declarative, AgentSpeak(PL) is able to reason about declarative goals and create plans which, when executed, ensure that a certain world-state is true. The approach taken in AgentSpeak(PL) consists of evaluating the consequences of procedural plans in terms of belief additions and deletions and converting these plans into a STRIPS-like representation, which can then be supplied to a classical planner along with the current belief base and the desired goal state.

An early example of the second approach is Ferguson's TouringMachine architecture [82] , whose reactive layer is controlled by a HTN planner. Such a organization preserves the BDI ability of using plans with black-box, opaque functionalities, such as continuous sensing-acting behaviours. However, the two-layered approach in InterRaP is inherently exposed to need of performing frequent re-planning in dynamic environments, as it does not take advantage of the ability of the BDI

system to constrain the options available to the agent and manage the long-term goals and intentions of an agent.

Efforts in creating a balance between deliberation and reactivity involving temporal planners include [44] and [22] [17] - both using an agent-system to monitor plan execution.

In [44] , the CLAIM agent architecture is applied to a robot system and extended with (i) a temporal planning component, and (ii) an execution monitoring and plan repairing component. The planner is executed every time the robot is requested to achieve a new goal. The resulting total ordered plans are converted into temporal plans by extracting from the domain knowledge base all the information about the durations of all the actions and offsets of all the effects. The agent-based executor system takes care of executing the actions at their planned timestamps after waiting for the termination of eventual prerequisite actions. It also checks for any discrepancy among the current world state and the one anticipated by the Planner for the execution of these actions, in which case the a plan repairing component is activated.

The ability to execute plans autonomously and to react without needing a new plan to be uploaded from Earth is a priority for future rover exploration missions. The work in [22] and [17] describes an autonomous control system obtained by integrating the IxTeT temporal planner and OpenPRS on top of the LAAS robot control architecture (see Figure 3). IxTeT is a temporal constraint-based causal link planner. It uses CSP techniques to maintain the consistency of the plan constraints. In particular, the planner is using a Simple Temporal Network for the temporal constraint. The plan is represented by state variables ranging over finite symbolic and numerical domains. These plans use chronicles to describe the world, its evolution and the planning problem. The explicit representation of the time permits to have temporally extended goals, durative actions and rendez-vous or visibility windows.
IxTeT has been integrated in the decisional level and interacts with the user and the robot's functional level through a procedural executive (OpenPRS). First, IxTeT produces a plan to achieve a set of goals provided by the user. The plan execution is controlled by both procedural and temporal executives as follows. The temporal executive decides when to start or stop an action in the plan and handles plan adaptations. OpenPRS expands and refines the action into commands to the functional level, monitors its execution and can recover from specific failures. During execution, OpenPRS reports any action failures and any possibly discrepancy between planned and actual resource consumption to the planner, in order to re-plan or repair the plan. As several IxTeT actions can be performed concurrently, it has also to schedule sequences of refined actions. It finally reports to IxTeT upon the action completion.

*The RUBICON's Control Layer will implement an hybrid of these agent-planning integration techniques and will apply them to its multiagent organization (thus supporting both R3.PROACTIVITY and R3.SCALABILITY). WP3 will extend state of the art integration between agent and temporal planning in order to take into account the contribution of the RUBICON's Learning and Cognitive layers (thus supporting R3.ADAPTATION)***.**

3.2.5 Learning in BDI Systems

The goal of agent and multi-agent learning is to gradually improve the behaviour of a single and multiagent systems (MAS) in order to adapt it to the environment and its changes over time [73] [74]

Results from work exploring modular learning solutions [76] are also applicable to BDI agent systems and are particularly relevant to the RUBICON project, as these approaches avoid to addressing the curse of dimensionality associated to learning sensing->action behaviours and focus on learning how to co-ordinate existing agent's skills.

In particular, finding which goals must be pursued, when they should be dropped, and which plans should be used to achieve them, are the fundamental problems tackled by applying machine learning techniques to BDI-based systems.

Within RUBICON the first two problems fall into the responsibilities of the Cognitive Layer while plan selection is the responsibility of the Control Layer - although with the help of the Learning and the Cognitive Layer.

As outlined in Section 3.2.1, the first step contributing to plan selection in BDI systems is the evaluation of the context conditions associated to each plan. These conditions state the conditions under which the plan is a sensible strategy to address the corresponding goal in a given situation. The execution of a BDI system relies then entirely on *context sensitive subgoal expansion*, allowing agents to “act as they go” by making *plan choices* at each level of abstraction with respect to the current situation. Section 3.2.1 has discussed how such an execution style leads to a goal-plan execution tree.

The fact that the plans' context conditions are restricted to be boolean formulas fixed at design time has important implications for the whole BDI approach, respectively:

1. It is often difficult or impossible for the programmer to craft the *exact* conditions under which a plan would succeed.
2. Once deployed, the plan selection mechanism is fixed and may neither adapt to potential variations of different environments, nor take in account the experiences of the agent.
3. Since plan execution often involves interaction with a *partially observable* external world, it is desirable to measure success in terms of probabilities rather than crisp logic, boolean values.

Some of the methods used for learning in BDI systems include the use of procedural learning to retrieve plans from collaborative agents [77] [78] , and integration with Case Base Reasoning [79] .

Alejandro et. al, [77] describe a BDI agent architecture extended with top-down induction of decision trees (TDIDT) used to learn when plans are successfully executable. Decisions trees are a widely used and efficient machine learning technique to approximate discrete value-target functions. Learned functions are represented as trees, corresponding to a disjunction of conjunctions of constrains on the attribute values of the instances. For each plan, the system generates a log of training examples for the learning task. Items to build these examples include: the beliefs characterizing the moment when the plan was selected, the label of success or failure after the execution of the plan, the plan-id, etc.

Experiments show the feasibility of such an approach and emphasised how BDI agents situated in a MAS increase their chances of learning if they can share training examples. However, it has been shown [81] how it can be problematic for decision trees to assume a mistake at a higher level in the goal-plan hierarchy, when a poor outcome may have been related to a mistake in selection further down. The point is that it is conceivable that the failure of a sub-goal could have been avoided, had an alternative plan been chosen earlier in the execution of the goal-plan hierarchy.

The solution provided in [1] consists in a probabilistic plan selection mechanism that caters for both exploration and exploitation of plans by using a *confidence* measure based on how much the agent has explored the space of possible executions of a given plan. The more this space has been "covered" by previous executions, the more the agent "trusts" the estimation of success provided by the plan's decision tree.

The judgement as to whether plan choices were sufficiently "well informed," is however not a trivial one. A failed plan *P* is considered to be *stable* for a particular world state *w* if the rate of success of *P* in *w* is changing below a certain threshold. In such a case, the agent can start to build confidence about the applicability level of *P*. The stability notion extends to goals as follows: a failed goal is considered *stable* for world state *w* if all its relevant plans are stable for *w*. When a goal is stable, the plan selection for such goal is regarded as a "well informed" one.

RUBICON's Control Layer will leverage the Learning Layer to estimate the probability of successfully execution of a plan in a given context. The Control Layer's two-tier multiagent organization will ease the implementation of a hierarchical learning approach. Section 4.5.3 will examine how the use of pre-conditions of temporal planning rules can help to decide how to attribute success and failure of plan execution. This will provide the basis to tackle R3.ADAPTATION.

3.2.6 The NUID UCD BDI Agent Framework

The Agent Factory Framework developed at NUID UCD is an open source collection of tools, platforms, and languages that support the development and deployment of multi-agent systems. The framework is broadly split into two parts: support for deploying agents on laptops, desktops, and servers; and support for deploying agents on constrained devices such as mobile phones and embedded devices, such as Java-enabled wireless sensors network nodes. The former support is realised through Agent Factory Standard Edition (AFSE) [7] , and the latter support is realised through Agent Factory Micro Edition (AFME) [25] . Both editions are open source projects and are freely available for download from the Agent Factory SourceForge web site[3] under the terms of the GNU Lesser General Public License.

While AFSE is based on the standard Java platform, AFME is based on the Constrained Limited Device Configuration (CLDC) Java platform augmented with the Mobile Information Device Profile (MIDP) rather than J2SE. CLDC and MIDP constitute a subset of the Java 2 Micro Edition (J2ME) Java specification.

Although there are significant differences in the infrastructure used to build the platforms, both AFSE and AFME are consistent in terms of their support for executing agents written in the Agent Factory Agent Programming Language (AFAPL). Communication on both systems is FIPA[4] compliant and thus interoperable. Agents on an AFME platform can migrate to a standard platform and vice versa. This consistency enables the developers of AFME applications to use the pre-existing integrated development environment, methodology, and compiler for the creation of agent designs for constrained devices.

AF models agents as mental entities whose internal state consists of beliefs and commitments. Informally, beliefs represent the agent's current state of its environment, while commitments

[3] http://sourceforge.net/projects/agentfactory/
[4] http://www.fipa.org

represent the outcome of an underlying reasoning process through which the agent selects what activities it should perform. In AF, an agent has both primitive abilities, in the form of directly executable actions, and composite abilities, in the form of plans built from control operators such as SEQ (sequential execution), OR/XOR (branching), and FOREACH (loop). Execution of an AF program involves the update of the agent's mental state by repeatedly applying an internal reasoning process that combines: update of the agents beliefs via perception of the environment through a set of auxiliary Java components, known as *perceptors*; the adoption of new commitments though the evaluation of a set of commitment rules, which map belief states onto commitments that should be adopted should that state arise; and the realisation of commitments by performing actions, which are implemented through a set of auxiliary Java components, known as *actuators*.

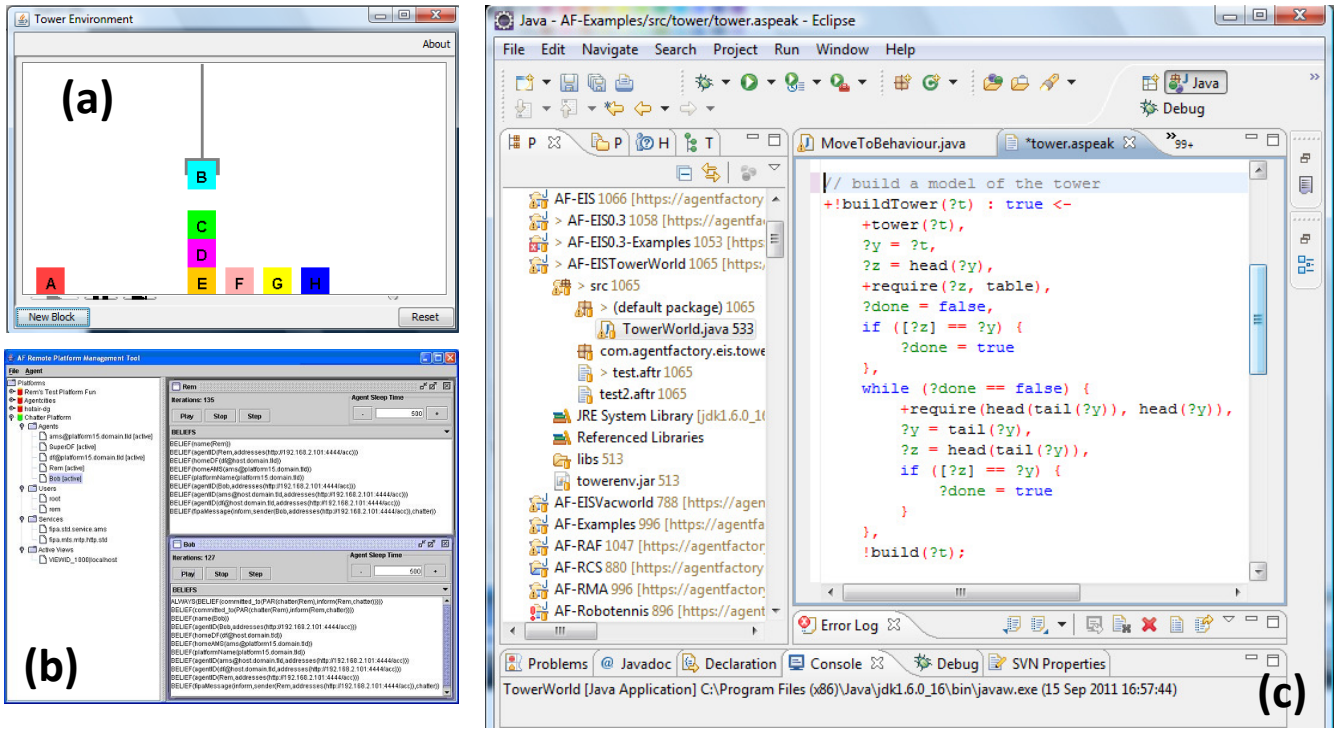

Figure 6. Screenshots from (a) the the TowerBlock test-bed, (b) the AFSE Agent Monitoring and Debugging Tool, and (c) the Eclipse AF Plugin

Similar to Jade [45] , AFSE structures the execution layer by distinguishing between agents and agent platform, which then provides the functional base upon which agents can operate in their environment. Also similarly to Jade, AFSE is not tightly coupled to a specific agent programming language.

The key features of AFSE are:

- **Run-Time Environment (RTE):** A FIPA compliant layer that consists of a set of modular and extensible agent platforms that can be easily customised for different applications domains[5].

[5] http://www.agentfactory.com/index.php/Run-Time_Environment

- **Common Language Framework:** A collection of components that facilitates the design and implementation of diverse Agent Programming Languages, including: a generic logic framework with forwards and backwards chaining; a plan execution framework; a common API model based on sensors, actions, and modules; an outline Grammar and template compiler implementation based on JavaCC; and a configurable debugging tool (see **Error! Reference source not found.**).
- **Eclipse Integration:** AFSE is fully integrated with the Eclipse in a way that simplifies the task of providing support for new languages and architectures[6].
- **EIS-Enabled:** The Environment Interface Standard [48] [7] is a standard for linking agents to application environments and test-beds. Through EIS, AFSE provides a number of test-beds where to test its agent languages.

Figure 6**Error! Reference source not found.** shows one of the plans used to solve the Tower World problem, being developed in the Ecplise IDE with the Agent Factory plugin.

NUID UCD's pre-existing tools will be used to facilitate the creation of robust and efficient plan execution and monitoring functionalities. In particular, the AF multi-language framework will be used to create a new agent programming language to handle the integration with a temporal planner (thus supporting R3.ROBUSTNESS, R3. RELIABILITY, R3.EXECUTION & CONFIGURATION, R3.PROACTIVITY, and R3.MONITORING). NUID UCD's expertise with embedded agent solutions for computational constrained devices will help the creation of agent solutions for WSN nodes (thus supporting R3.HETEROGENEITY, and - by enabling the direct interaction with the RUBICON's Learning Layer - also R3.ADAPTATION and R3.SCALABILITY).

[6] http://www.agentfactory.com/index.php/Agent_Factory_Eclipse_Plugin
[7] http://www.agentfactory.com/index.php/EIS

4. High-Level Design

4.1 Objectives

Much of the research into ubiquitous and networked robotic systems has focused on the construction of environments composed of simple devices that together can accomplish complex tasks. This is generally done by abandoning the traditional view of autonomous robotics (which tends to build multi-purpose, often humanoid robots) in favour of achieving complex functionality by composition of distributed (and specific-purpose) sensors and actuators.

Unlike the traditional robotics approach, in which learning and adaptability have long been a hot research topic, research within ubiquitous robotics has so far relied mostly on *symbolic* and *static* models of perception, reasoning and goal deliberation – especially in the context of *distributed* and *decentralized* systems such as the ones considered for the two application domains of RUBICON.

The two key goals of the Control Layer within RUBICON are to provide sensing and actuating services that are adaptable and robust.

The first of these two goals requires that arbitrary combinations of a subset of heterogeneous devices developed for RUBICON should be able to be deployed in unstructured environments such as those exemplified in the AAL and hospital transportation application scenarios. Our aim is to perform tasks through the combined use of heterogeneous devices. In order to decide the specific behaviours which, in combination, achieve necessary and meaningful tasks, we require a *general* high level control mechanism. This mechanism should not be restricted to only those situations that are envisioned by the designer of the RUBICON system, rather the high-level controller must exhibit the ability to adapt to varying environmental conditions and requirements.

In addition to adaptability, we require that the overall control mechanism is also capable of synthesizing robust strategies, in the sense that these strategies should take into account both a sufficient amount of exogenous events and the specific capabilities of the devices used to enact the strategies. Unfortunately, the robustness stands in contrast with computational tractability Specifically, robust behaviour of the ecology contrasts with the combinatorial explosion stemming from the many requirements that must be upheld during strategy synthesis and execution. Another important dimension to consider is that there are many ways to achieve objectives within a specific application scenario. For instance, a robot may be able to localize itself with the help of an environmental camera or through the use of an on-board laser range-finder.

The key technological gaps that need to be addressed by any solution in order to satisfy the requirements of the Control Layer within RUBICON are:

1. The solution must be able to perform any reasonable task that can be achieved by a combination of the capabilities of any of the individual devices available in ecology.

2. It must be deployable in novel environments without requiring overly-complex a-priori configuration.

3. It should be robust to failures of individual devices and exhibit graceful degradation properties.

4. It should be capable of using alternative means to accomplish goals when multiple courses of actions are available.

5. Furthermore, it must be able to act on goals, as given by the Cognitive Layer, within a tractable time frame (scalability) and to adapt to dynamically changing goals with implicit time constraints on deliberation.

We note that (3) stands in contrast to the use of application-specific models since we cannot expect end-users to write such models (2). Furthermore, very complex models that try to encapsulate every possible situation that may arise lead to intractable planning problems, thus contradicting points (4) and (5) above.

We propose to develop a system that adaptively learns the application-specific models, and integrates these models with general purpose planning strategies. We claim that this will ameliorate the issues arising from the points above. Thus we can specialize our challenges as follows:

1. The system must be able to learn how to combine devices, and dynamically re-learn these combinations as devices are introduced/removed from the environment.

2. Due to the combinatorial growth of possible execution traces (which constitute the training data for learning processes), the learning problem must be structured and limited so that the system can adapt to changes within a tractable time frame (both in terms of computational speed as well as number of situations encountered by the system).

3. Due to the distributed nature of RUBICON ecologies, learning would not be performed in the same physical location as deliberation, therefore creating bottlenecks in communication bandwidth.

The first point can be solved by using decentralized learning methods. To address point (2), we intend to rely partly on a general purpose model description of the RUBICON ecology that provide the Learning Layer with structural information about the different learning tasks.

In summary, the Control Layer will:

- formulate and execute both action and configuration strategies to satisfy the goals set by the Cognitive Layer and the necessary collaborations set for each node.
- exploit the RUBICON Learning Layer to improve its perception capabilities and adapt these strategies to the perceived situation of the environment and the system, and to improve the quality of the same strategies over time.
- put in place a vocabulary that clearly describes the capabilities of each component, and introspection and coordination capabilities to carry out actions and configuration plans with heterogeneous and open (with components leaving and joining the system) scenarios.

4.2 Multiagent Approach

The key to tackling the above issues in our architecture (illustrated in the next section) is to move away from the centralized solution implemented in most typical ubiquitous robotics solutions (such as the *Ecology of PEIS* project). Instead, we frame the Control Layer as a multiagent system (MAS).

A MAS-based solution is motivated by the following considerations:

- ***Adaptation by close interaction with the Learning Network*** - A MAS is best situated to exploit the RUBICON's learning capabilities in order to refine both agents' perceptions and their control strategies over time, as the Learning Network described in D2.1 is scattered across a network of heterogeneous devices constituting the RUBICON ecology.

- ***Scalability*** - Scalability can be promoted by separating global, high-level concerns, possibly involving multiple participants to the RUBICON ecology, from the low-level concerns specific to each individual participant.

- ***Robustness and Reliability*** - Granting more autonomy to each participant of the RUBICON ecology, rather than subjecting every participant to the control of a centralized planner, enables the RUBICON Control Architecture in each participant to find and carry out more complex, and thus more robust and reliable action and configuration strategies without overloading a central control point.

- **Knowledge Sharing for Heterogeneity & Open Environments** - A MAS-based solution provides individual agents with coordination and introspection capabilities. These capabilities directly enable dynamic composition of primitive capabilities in the pursuit of high-level goals. In addition, the problem of understanding which goals are relevant is de-coupled with the problem of understanding how to achieve them, therefore supporting the ability to share and apply acquired knowledge within heterogeneous and open RUBICON ecologies.

- ***Computational Constraints*** - Agent technologies, such as those implemented by NUID UCD, provide a number of options for their integration onto computationally constrained devices and the exploitation of the RUBICON's Communication Layer.

4.3 System Architecture

4.3.1 Overview

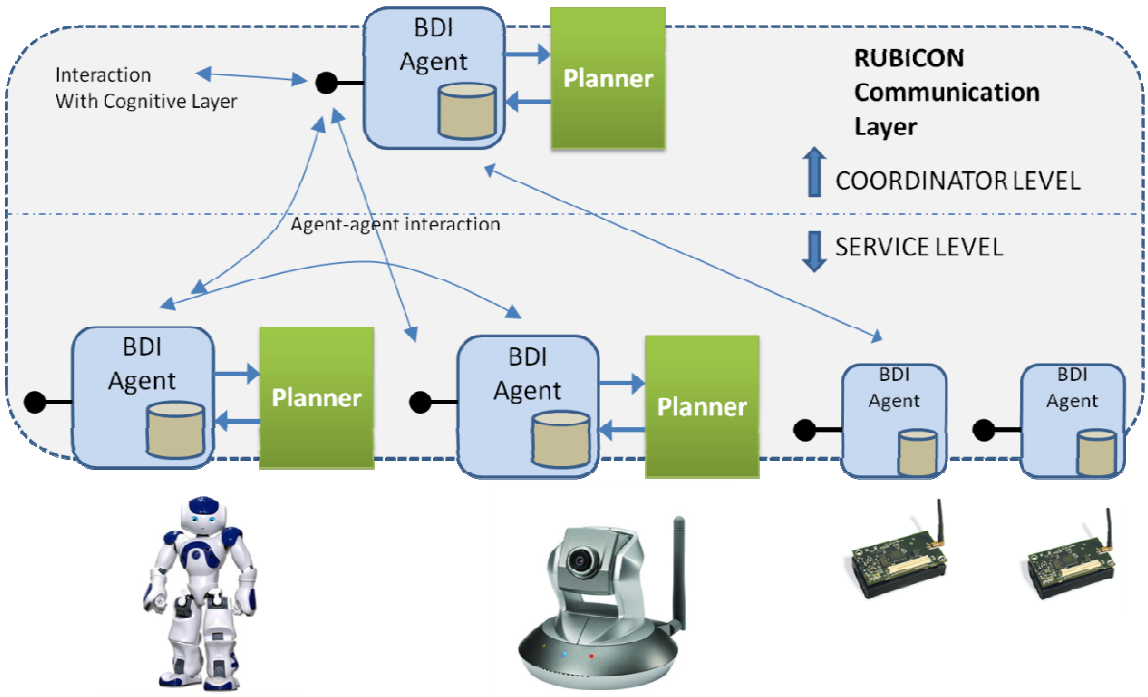

Figure 7. High-Level system architecture of Control Layer framed as multiagent system

We present in this section an overview of the high-level design for the control layer developed as task T3.1 and go into further details in the following sections.

Figure 7 illustrates the design of the distributed RUBICON Control Layer made up of multiple control agents combining both BDI and planning components.

The proposed architecture aims to implement an hybrid between the two main agent/planner integration strategies examined in Section 3.2.4 and apply it to a MAS hierarchically partitioned into two levels, respectively: (i) the **Coordination** Level, and (ii) the **Service Level.**

The coordination level is populated by one or more agents in charge of the coordination of multiple agents in the service level (Figure 7 depicts only one such coordination agent, for simplicity). Coordination agents are concerned with high-level goals and their decomposition in sub-goals to be achieved in a co-ordinated fashion by multiple agents operating in the service level.

Agents in the service level are logically associated to robots, actuators, appliances, and WSAN nodes, and are in charge of controlling the action and configuration strategies carried out by these devices, as well as their interaction with the larger system. For computationally constrained devices, such as WSAN nodes, this will be achieved by employing small footprint agent solutions without planning

capabilities, which will interact with proxy and coordination agents (running on more powerful devices) through the network, as illustrated in more detail in Section 4.7.2.

Planning and plan execution at the coordination level are simplified as they do not have to account for all the details specific to the inner workings of each agent operating in the service level. Rather, agents in the coordination level deal with more abstract representations of capabilities. They delegate to service agents the control of the actual action and configuration strategy needed for execution. On a system scale, this resembles the hierarchical organization used in single-agent/planning architectures, such as those implemented in [2] and [22] . At the same time, planning within each agent is also simplified, as it is supervised by the corresponding BDI agent, which helps to focus the planning activity toward a set of specific goals (the agent's commitments), as in traditional BDI/planning integration approaches [15] .

The other main motivation behind the adoption of such an high-level architecture is to ease the integration with the RUBICON's Learning layer. Service agents will leverage the Learning Layer to improve their perception capabilities and gather accurate and predictive information about their situation and the status of the environment, and also to adapt their action decision process based on their past experiences. Coordinator agents will leverage the Learning Layer to devise and supervise the execution of plans for the whole RUBICON ecology, and also to manage the sharing and re-use of what is learnt by the agents in the Service Level.

Employing coordination agent allows us to introduce a "controlled" MAS dynamic compared to a "full" MAS made out of loosely coupled agents with no hierarchical organisation and engaged in pure multiagent planning. In the first instance, we foresee the use of a single coordinator agent to be used as the front-end interface toward the cognitive layer, to deal with the goals set for the entire RUBICON ecology. However, systems with multiple coordination agents may be possible, to provide horizontal partitioning of concerns within the coordination level. For instance, in the AAL scenario, there may be a coordinator dealing with concerns such as the temperature of the air and the water, and a co-ordinator dealing with safety concerns, both requiring the operations of agents operating over the home automation systems. It may be convenient to separate these concerns, as they can be satisfied by largely disconnected set of entities, and leave the respective coordinators to resolve the occasional inconsistency, for example, arising from a conflict in resource usage.

Another use for multiple coordination agents occur in larger ecologies where conflicting goals may be formed under the authority of different users. Consider a scenario where two families inhabiting the same apartment building have deployed a RUBICON ecology in their respective apartments. Each such ecology would house its own coordination agent with their own service robots and devices dedicated for the use of the families only. However, they may also (or the landlord may) choose to install shared devices that are capable of for instance monitoring the entrance to the apartment building or actuate the temperature in common areas. As such there the whole set of devices need to be able to communicate, but with a partitioning where the private service devices does not respond to requests from any external coordination agent.

The proposed architecture does not pose any constraints on the organization and the deployment distribution and strategies used by the cognitive layer and the learning layer, as each agent can be connected to each of these layers (and to any other agent in the MAS) via the communication layer developed in WP1 (described in D1.1) by combining the PEIS tuplespace over the Ethernet network with the WSAN infrastructure.

The same architecture does not prevent (e.g. in future instances) the cognitive Layer to deal directly with the agents in the service level, as these may autonomously decide that they are capable of carrying out required tasks with limited or no supervision from the coordinator(s). The cognitive layer - or instances of it - can post goals to the system's tuplespace and be unaware of the agents (coordinator or service) that will first deal with these goals. In contrast, interaction between control and learning layer is peer-to-peer, with one instance of the control layer (agent/planner) using one instance of the learning layer to refine its own perception capabilities and inform its action and configuration strategies. Such an interaction requires a prompt exchange of information between learning and control layers. This can be afforded without much problems within the WiFi ethernet LAN. However, for WSAN nodes, it is envisaged that the control layer will be deployed within the same host of the corresponding learning layer, in order to limit communication over the WSAN, which would negatively impact on both energy and bandwidth consumption.

4.3.2 Multiagent Interaction

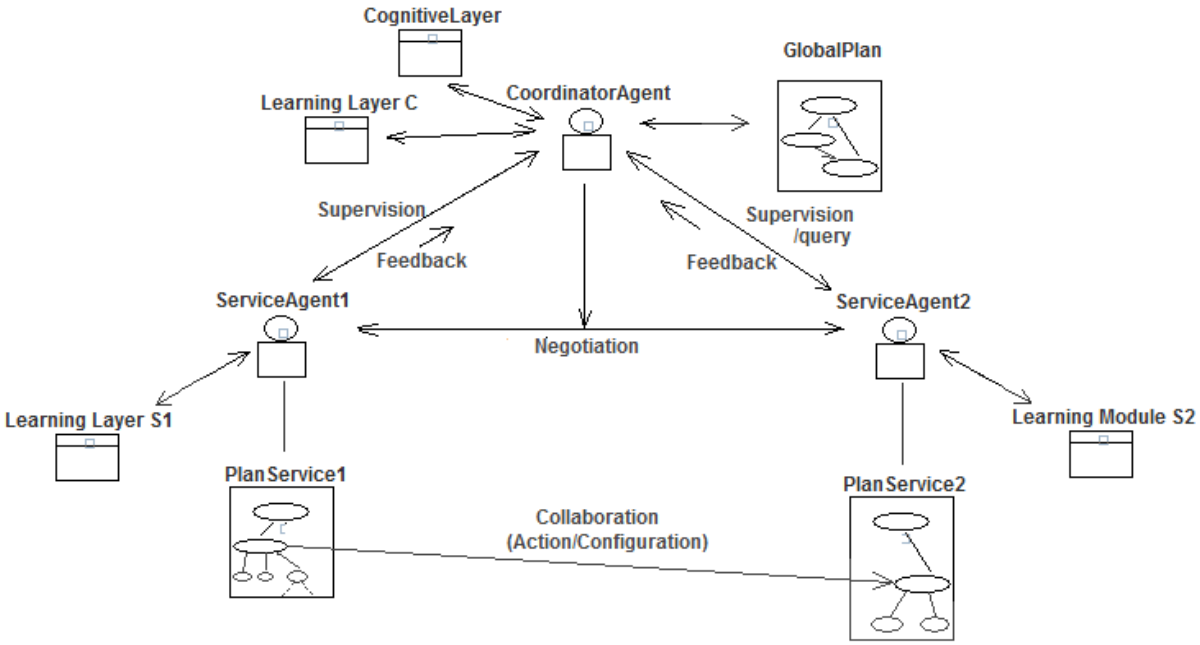

Figure 8. Multiagent Interaction within the Control Layer's System Architecture.

Figure 8 helps to clarify the interaction envisaged between the agents in the Control Layer and the other layers in the RUBICON's architecture (Learning and Cognitive).

At least in the first instance, one single coordination agent (the “coordinator agent”) will act as front-end interface toward the Cognitive Layer, from which it will receive goals set for the whole RUBICON ecology. The coordinator agent will be left to find and supervise the execution of a number of high-level, global plans, in order to achieve as many goals as possible given the resources available in the Service Level (robots, sensors...), and the status of the environment.

Each of these global plans may require multiple service agents, to pursue possibly dependent subtasks each contributing to their execution. These subtasks may describe partial situations (sub-

goals) to be achieved on the way of solving the goals set by the Cognitive Layer. For instance, if the Cognitive Layer asks to inform the user of a fire hazard, one of the sub-goals to be achieved will be to have one robot with a speech synthesiser in the same room with the user. In addition, sub-goals may also refer to required configurations of the RUBICON Ecology. For instance, the decision to move a robot toward a specific location may require the ability to track the position of the robot during its movements. Both type of service-service interactions can be captured by dependencies (temporal, logical) between actions and/or sub-goals in the global plan.

In order to provide a suitable plan for the RUBICON ecology, the coordinator agent will need to maintain an up-to-date picture of (i) the available service agents, (ii) their high-level capabilities (i.e. the goals they are capable to achieve), and (iii) their high-level situation, including their location and status update about their current activities.

Dedicated communication protocols will be used to coordinate this exchange of information.

The service agent themselves will notify their coordinator agent with their status feedback. Due precautions should be taken in order to reduce the bandwidth used for status updates. For instance, since the coordinator agent is not concerned with low-level details, such as navigation, it should be sufficient for each service agent associated to a mobile robot to notify the room where these robots are located, or any way-point they have reached, rather than constantly sending their most up-to-date location estimate.

In order to update the coordinator of their capabilities, each service agent can transmit a list of simple goals it is able to handle (given its sensors and actuators hardware and software components). For instance, a robot may inform the coordinator, via the system's tuplespace, that it can be instructed, in theory, to move into any given room, and that it can provide estimates of its own position.

With this information, the coordinator agent will be able to find a number of high-level plans, each equally suitable to achieve the cognitive goals. It will then exploit the Learning Layer to inform the selection of some of these options. In addition, a negotiation rounds will be used to finally identify one single plan out of multiple remaining options, and also optimize the way it is distributed across the agents in the Service Level. During the negotiation (e.g. market based), the coordinator will issue requests the service agents to bid for being allocated specific subtasks in the global plan. At this stage, each service agent will need to leverage what is learnt by the Learning Layer to assess the actual confidence (rather than a crisp Boolean value) they have to be able to successfully satisfy the coordinator's request goal given the current perceived situation of the RUBICON ecology and the environment. This will allow a modular solution to take in account service-level knowledge without having to incorporate it in the coordinator's reasoning process. For instance, a component in one service agent may fail thus invalidating all the capabilities that depended from it, or a robot may learn that it is not able to safely navigate through a specific room. Through negotiations, these self-assessments will be performed at the Service Level, and communicated to the coordinator only in a summary form, as bids used for negotiation.

Once subgoals have been assigned, each service agent will be responsible to refine them, for instance, by finding their own plans to decide how they are going to use their acting and sensing capabilities to carry out these subgoals. At this stage, dependencies in the global plan will translate in collaborations set between two or more service agents. In addition, service agents may autonomously decide to set further collaborations, or to modify the ones already set by the

coordinator, as long as these interventions do not interfere with the global plan. For instance, a robot instructed to reach the kitchen by using its laser-based self-localization capabilities may safely decide to start localizing by collaborating with a camera mounted on the ceiling of the apartment, once it detects that its self-localization sub-system is performing poorly, as the new collaboration will not impact on mutually exclusive resources (if we do not consider the minimum impact on consumption of WiFi bandwidth).

In addition, the coordination agent will keep monitoring the status updates sent by the service agents, and decide if the RUBICON ecology needs a new plan, for instance, to account for new goals, but also to repair/modify the current plan. Plan modification may occur in the case of failure in any of the sub-goals currently being pursued by the service agents, or in case a service agents is not making enough progress, for instance, because a robot is finding difficult to navigate to a specific location, or because there is not enough light to use the camera-based localization system.

4.4 Interaction with the Cognitive Layer

The driving source of all tasks of the Control Layer in the RUBICON ecology originates in the goals provided by the Cognitive Layer and the subsequent subtask that are needed to satisfy them and/or to maintain previously specified goals as the environment evolves during the execution of tasks.

The coordinator agent in a RUBICON ecology (initially we consider only the presence of one such agent, and plan to expand with multiple coordinator agents in future iterations) is responsible for maintaining a list of *prioritized current goals* that are given by the Cognitive Layer. While the format and details of these goals are specified in detail in deliverable D4.1, here we give a brief reminder of them:

- The Cognitive Layer provides a set of goals (by posting goal tuples to the tuplespace*)* where each goal contains
    - Goal type identifier, which references a list of pre-determined goal predicates.
    - A parameter vector, specifying the arguments to the goal predicates with a pre-determined semantic ordering of the vector arguments.
    - Priority of the goals
    - Time of issue, deadline by which goal must be satisfied and earliest allowed starting time.

  In addition, the Cognitive Layer already employs cognitive reasoning intended to avoid posting goals that are logically inconsistent with each other (e.g. *open window during daytime & keep room in the dark*, or *achieve temperature set point of 24 °C & maintain temperature between 18 and 22 °C*, in the AAL scenario) or clearly leading to resource conflicts (e.g. *load trolley1 with robotA, load trolley2 with robotA*, in the transport scenario).

- The Control Layer shall attempt to satisfy all goals within their respective deadlines and shall consider plans satisfying higher prioritized goals before goals with lower priorities when not all goals can be satisfied simultaneously.

- Goals can be removed by deleting the corresponding tuple giving that goal. Note that deleting a goal is not the same as asserting it's negative. E.g. If a goal is posted to turn off the light before timepoint *T*, and the goal is removed before that timepoint it is still not known whether the lamp is on or off since the Control Layer may have turned it off just before the goal deletion. Only by asserting that the light should be on can we ensure this state.

- New goals can be posted and old goals can change properties at any time.

- The Control Layer shall provide status feedback for the goals that are currently planned to be achieved and the ones that cannot currently be achieved.

The Control Layer will subscribe to all goal tuples provided by the Cognitive Layer and will monitor them for changed properties. At least initially, this is done only on the level of the coordinator agent(s).

Whenever the set of current goals change these coordinator agents will translate the set of goals into a *goal description* together with a *domain description* that is suitable for the coordinator planner. The translation to goal descriptions is performed trivially through the list of agreed upon predicates for corresponding to each goal and by posing the temporal constraint as given.

In addition, the Control Layer will monitor the performance of its plans, and any events arising during their execution (such as way-point reached, major resources used, etc.), to prepare summary and interim feedbacks to the Cognitive Layer (in form of tuples corresponding to the R4->3.STATUS and R4.EXPLORATIONS requirements).

4.5 Interaction with the Learning Layer

The specific policy that will be used by the planning algorithm to create plans and leverage the RUBICON's Learning Layer are the objects of future work within this project.

The following sections explain the directions we will investigate to support such an integration.

4.5.1 Planning Domain

Control agents both at the coordinator and the service level will receive from the Learning Layer updated assessments of the alternative plans deduced by the controller. These evaluations are context-dependent, and are at the granularity of the specific actions that the plans prescribe.

The *domain description* consists of a *predefined high level model* that specifies the capabilities of each device in the RUBICON ecology. The available operators that specify what each device can do is read from the tuplespace of the corresponding BDI agent of the device. The high level model and the operators are written *optimistically* and does not attempt to capture all the nuances of the unstructured environment in which they are operating.

The planner is invoked multiple times by the BDI agent. For each operator that is given by the devices and that appears in one of the generated plans, a possibility of success is given (originating in the Learning Layer) for each possible instantiation.

The way in which the Control Layer and the Learning Layer operate together is best understood with an example. Suppose that the Cognitive Layer has identified as a goal (i.e., a state of the environment and its actuators) that the robot should be in the kitchen (e.g., to monitor the state of the water tap). In order to reach the kitchen, the robot needs to be localized. Within a Robotic Ecology, robot localization is a task that can be achieved through various means, including the use of a laser-based self-localization algorithm, or the use of a ceiling camera in conjunction with artificial vision software. Crucially, though, for the latter to work there must be light in the kitchen while the robot is being localized. Sufficient light, in turn, can be achieved in two ways: on one hand, opening the blinds to let external light will achieve a sufficient level of luminosity; on the other, the light may be turned on. The planner knows that both these courses of action are possible, and it also knows the precise temporal relations that exist among these elements of the plan. Specifically, the planner's domain is the following:

1. in(room) DURING localize(room)

(To be in room R, the localization service in room R must be active)

2. localize(room) DURING light(room)

(The localization service in room R is provided when there is light in room R)

3. light(R) EQUALS lightSwitch(room,on)

(We have light, exactly when the lamp is on)

4. light(R) EQUALS blinds(room,open)

(We have light exactly when the blinds are open)

Given a domain like this, the planner will provide the BDI agent one of two alternative plans, namely (a) opening the blinds and then starting the localization service, or (b) turning on the light and then starting the localization service. The choice as to which plan is preferable is not necessarily known to the planner. For instance, it may have sufficient elements to choose plan (a) if the blinds are already open, i.e., justifying switching on the localization service by a fact that is already present. However, it is not true in general that the planner always knows which plan to prefer. Indeed, the aspects of the real world that are not modelled in the domain may very well determine the failure of a particular plan. In this example, these conditions may be that it is night, and therefore opening the blinds will not produce the sought after effect of having light in the kitchen. The rationale for not modelling this information in the domain is that it would add a significant burden to the modelling effort. This would ultimately impact both the performance of the planner and the effort required to deploy the system in different physical environments.

Although it is practically not possible to subsume all relevant information in the planner model, we nevertheless want to maintain the possibility to model those elements of the real world that are known to be relevant for our domain. In other words, if we know that a luminosity sensor will reliably report the conditions under which sufficient light can be achieved by opening the blinds, we should be able to do so. RUBICON strives to empower the deployer of the smart environment with the ability to choose the level of granularity at which domain modelling occurs. This is achieved by means of a tight integration with the Learning Layer. The interaction between control and Learning

Layer is mediated by BDI agents, and the fundamental mechanism by which *unforeseen* (i.e., not modelled) contextual information is take into account is through feedback from the Learning Layer.

4.5.2 Planning in the Control Layer

The domain consists of synchronizations. In the simple example above, there are four synchronizations, each describing the temporal relation that should exist between predicates modelling sensor readings (e.g. whether there is light in room R), actuators (e.g. the robot's mobility subsystem), or knowledge resulting from perception and other sub-systems (e.g. localization, object recognition).

The dependencies may be more complex than those shown in the example, as they model multiple dependencies. For instance, the synchronization

- in(room) { DURING localize(room), MET-BY charged, BEFORE [0,20] ¬in(room) }

models the fact that to be in room R, the robot should be localized, its batteries should be charged, and that it should not stay in a room more than 20 units of time.

Overall, the planner's domain contains predicates that model one of three types of facts:

- a desired state of an actuator component (e.g., blinds(R,open))
- a derived state that can be achieved (e.g., light(room), localize(room))
- a sensed state that can be verified

Predicates need not be of one type. In the example above, we have so far assumed that light(room) is a derived state, i.e., a state which is achievable by instantiating other synchronizations (hence, synchronizations 3 and 4). However, it is conceivable that whether there is sufficient light is a measurable quantity. If this is the case, and we have a sensor that can perceive luminosity and decide whether light(room) is true, we will say that light(room) is a predicate that represents a desired state that can be verified. This is trivially true for actuators, which most likely can be queried regarding their current state. Whether a predicate is of one, two or all three types, is determined by the *interfaces* this predicate has with the ecology. Specifically:

- A predicate designating a desired state of an actuator has an interface to the BDI agent, which in turn drives one or more physical actuator components; this interface monitors the assertion of the predicate in the planner's internal state space representation and synthesizes one or more commands for its actuators whenever the predicate is asserted.
- A predicate designating a derived state that can be achieved has no interfaces to the ecology.
- A predicate designating a sensed state that can be verified has an interface to the a physical sensor; this interface synthesizes and imposes into the planner's internal state space representation a predicate modelling the state currently sensed by the sensor.

Note that whether predicates represent one or another type of assertion does not affect the domain itself, rather what the planner does with the predicates during search. If the planner needs to support robot localization as in our example, and the light(room) predicate represents both a derived state and a desired sensed value, the planning algorithm has the option to either seek whether light(room) has been asserted by a sensing interface, or to pursue the achievement of this state by leveraging another synchronization. The former operation is called *unification*, while the latter is

called *expansion*. In our example, if the planner chooses to unify, then the produced plan will not contain any action involving the production of light, whereas if it chooses the latter, light will be produced either by opening the blinds or by flipping the light switch. Note that, while in this specific example attempting to unify is clearly the most desirable search strategy, this is not true in general, as “early” unifications may lead to dead ends later in the search and induce the need to backtrack.

The internal state space representation of the planner will be a constraint network, whose variables represent predicates as described above, and whose constraints are quantified temporal relations in Allen's Interval Algebra. Sensor interfaces write to this constraint network by adding variables and constraints modelling the particular sensed value as well as its placement in time, whereas actuator interfaces read the current variables and assess based on the temporal constraints when the corresponding command should be dispatched. At any point in time, the internal constraint network maintained by the planning algorithm can be viewed as a collection of *timelines*, i.e., functions of time representing the values of predicates.

4.5.3 Refining the Planning Domain and Informing Plan Selection

Initially, the domain description used by the planner contains a high-level specification of how the environment should behave in response to certain situations. Specifically, the domain is a collection of crisp "rules", called synchronizations. A synchronization establishes flexible temporal relations between sensor readings (which can include inferred context as it is provided by the Cognitive Layer) and actuation (i.e., plans that provide contextualized assistance to the user).

The specification provided in the planner's domain is brittle, but provides sufficient information to obtain one or more plans that can achieve the goals put forth by the Cognitive Layer. For the sake of simplicity, let us go back to the example domain introduced above, whereby the planner can produce two plans (a, b) to reach the kitchen. The two plans are evaluated by the BDI agent, which initially has no criteria to prefer one plan to the other. When executed, a plan will incur in either success or failure. The BDI agent will take note of the successes and failures in plan execution. In our example, two several situations are possible, one of which is that plan (a) is chosen and it is day (therefore opening the blinds provides enough light). In this situation, the agent will reinforce “score” of plan (a), which, as a whole, has achieved the desired goal.

Another possibility is that plan (b) is chosen – and the agent will consequently award a positive reinforcement to plan (b). If instead plan (a) is chosen at night, the agent will add a negative reinforcement to the plan. If the light(room) predicate is connected to a sensing interface, the negative reinforcement will not be attached to all elements of the plan, rather *only to synchronization 4* in the domain. The agent knows that this is the culprit of the failure because the light(R) predicate (which is the “head” of the synchronization) is also a sensor that can be queried for verification. If on the other hand we do not have a sensing interface for the light(R) predicate, then there is no possibility to know which synchronization should be punished/rewarded after execution.

Depending on the granularity of the detected failures, we may however infer which of the three active synchronizations should be trained. If the localize(room) functionality fails, then we know that synchronization 2 or 4 should be trained – but not synchronization 1. Assuming that we train only to a small degree with each example (e.g., decreasing the reinforcement of synchronizations 2 and 4 by 0.1), we can run another plan involving synchronizations 1 to 3 in the future and see whether it succeeds. In case it does, we can then increase synchronization 2 by 0.1 again, which in the long would indicate that plans involving synchronizations <1,2,3> but not those involving synchronizations <1,2,4> are suitable for the night time.

4.5.4 Enhance Perception Abilities

The other main contributions from the Learning Layer to the Control Layer will be the provision of both:

- reliable and accurate perception data, possibly obtained with the aid of multiple sensors and data processing sources, and

- reliable classification of (possibly) multiple events out of a predefined set of candidate events that might be occurring within the RUBICON Ecology.

An example of the first type of contribution is discussed in more details in Deliverable D2.1 - "4.4.1 Analysis of ESNs for learning in WSN applications", which presents preliminary results obtained by using learning solutions - of the type to be used within the Learning Layer - to timely predict the typology of user's movements in an environment comprising rooms in which a small WSN was installed. Within RUBICON, such a solution will be extended to learn to associate patterns of radio signal strength signal indicators (RSSI) received from a mobile WSN node (worn by the user and also installed on robots and other mobile devices), to location information, such as the room where the WSN node (and thus the user, or the robot carrying it) is located.

An example of the second type of contribution is the ability to provide timely and predictive information on the activities being performed by the user(s), such as recognizing that the user is cooking by analysing the signal and temporal pattern received from sensors installed in the kitchen, such as switches triggered upon opening and closing cupboards and refrigerators, RFIDs installed on bread-cutters, and energy monitoring solutions, such as those developed by NUID UCD to detect the usage of kitchen's appliances.

Both these contributions will improve the Control Layer's ability to discern the status of the RUBICON ecology and the environment. This will allow the Control Layer to devise situation-dependent, pro-active action and configuration strategies and also to account for contingencies and opportunities arising during the execution of these strategies.

However, for this to happen, the Learning Layer needs:

- to be informed about the existence of relationships between sensor data as well as between sensor data and user's activities. This can be performed by acting on the Learning Layer's wiring interface (discussed in Deliverable D2.1 "4.3 Specification of a Computational Task") to define a new learning task and specify the set of data sources relevant for the task (e.g. the set of identifiers of the input sensor transducers) and determines where the output predictions should be delivered (new outputs that the Control Layer will interpret, as location information inferred from RSSI patterns, or as assessments of user's activities).

- to be provided with training data, for instance, by presenting both RSSI pattern examples and location information gathered by driving a robot equipped with both a WSN node and self-localization abilities, and also assessments of the user's activities that can be used as teaching signal by the Learning Layer to learn to classify them autonomously, from raw sensor data.

The Cognitive Layer is expected to be the driving force under which the RUBICON ecology can discover dependencies between different sensor data, detect regularities between patterns of events (e.g. opening of the refrigerator, use of the bread-cutter) that can interpreted as indication of the

user being involved in some form of high-level activity (cooking), and also drive the Control Layer in a way that will maximize the opportunities to collect useful training data.

These cognitive abilities can be used to create and refine new learning tasks for the Learning Layer.

However, we also envisage that the Control Layer will cover the important role of supervisor for the Learning Layer. The rationale for this double-supervisory role is that the RUBICON ecology needs to provide a safe, reliable, and goal-oriented service environment for the user and that we cannot demand that the system has to learn everything from scratch, possibly producing erratic and unsafe behaviour in the first period after the RUBICON system is installed in a new environment.

To this end, the Control Layer will:

- facilitate the description of high-level synchronisation rules that can be used to initiate and "shape" what will be learnt by the Learning Layer.
- provide a means to exercise the Learning Layer independently from the Cognitive Layer, with clear advantages from a modular development perspective.
- open the way for enabling small footprint solutions leveraging simple control functionalities that do not require planning or cognitive functions, as discussed in Section 4.7.2.
- allow generalization support. The Control Layer is capable of "reasoning" on data at a higher, level, which means that it will be able to define new learning tasks based on high-level templates (stored in its domain ontology) that can be specialized once the Control Layer is informed of the exact resources (sensors, actuators, robots) that are available in the system

For instance, the Control Layer only needs a couple of rules (synchronizations) to represent the fact that there is a likely relationship between the location of a WSN node and the RSSI patterns the nodes receive while communicating with the remaining nodes in the RUBICON WSAN. Once the Control Layer discovers the actual topology of the WSAN, it can use this information to ask the Learning Layer to create a new learning task with as many inputs as the number of WSAN nodes discovered, and as many outputs to represent the location information (i.e., 3 outputs to represent room-level localization in an apartment with no more than 2^3 = 8 rooms). The Control Layer also knows that this output can be trained by using location information found by other means, e.g. via robot's self-localization, or by using camera-ceiling localization systems. As a result, the Control Layer will be able to apply high-level knowledge encoded in its domain to a variety of instances occurring in the actual environments where it is installed, and also cope with modifications occurring in these installations (both permanent and temporary).

The details of how we aim to leverage these capabilities to achieve our objectives can be better illustrated with a simple example.

A control synchronization can be used to establish flexible temporal relations between sensor readings, inferred context (e.g., human activities) and actuation (i.e., plans that provide contextualized assistance to the user). An example of a simple domain is shown in Figure 10, which shows two synchronizations that model how to recognize the activity of plan watering.

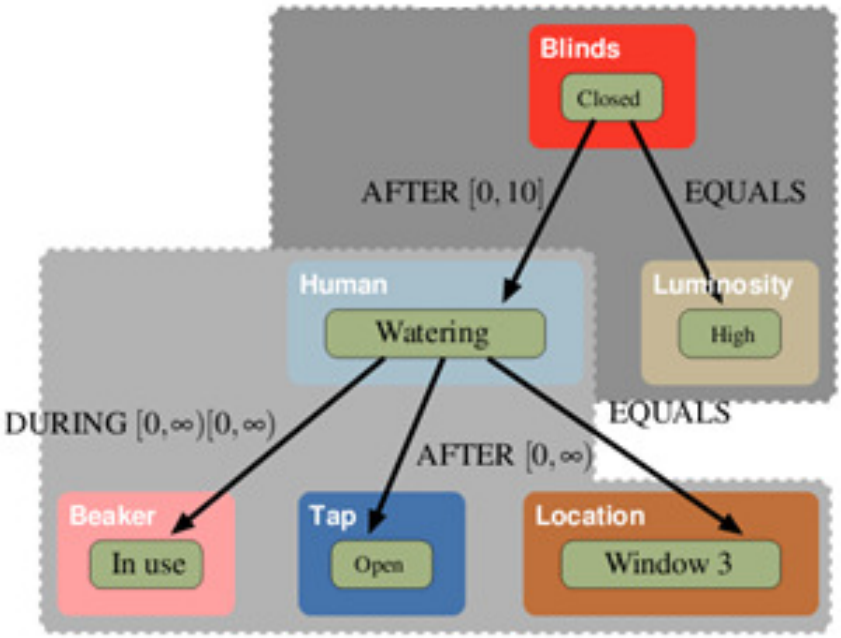

Figure 9. Example temporal constraint network representation of the task of watering a plant and the expected user behaviour of closing the blinds if also the sunlight level is high.

These synchronisations allow the system to recognize the watering activity only in nominal or close-to-nominal conditions. Specifically, watering will be recognized only if the timing of the sensor reading involved (beaker usage, tap opening, etc.) falls within the bounds prescribed by the temporal constraints. For instance, suppose that the tap is opened to fill the beaker, and that the sensor responsible for recognizing the flow of water from the tap delays recognizing that the tap is closed until after the user has reached the window to water the plant. This situation, although evidently an instance of plant watering, will not satisfy the constraints stated in the synchronization, as the constraints {(Watering EQUALS Window 3), (Watering AFTER Open)} should induce a temporal propagation failure.

For this reason, the idea in RUBICON is to gradually "delegate" elements of the domain the planner uses to the Learning Layer.

In order to transition from crisp and brittle descriptions to data-driven classifiers, we employ the initial domain specification to gather "evidence" of good system behaviour. More specifically, temporal sensor traces, contextual inferences and plans processed and inferred by the planner are logged during a "training" period. These logs will accumulate into what we can consider a rough set of training data in the form of timelines. An example of such timelines for the running plant watering example is given in Figure 10, below.

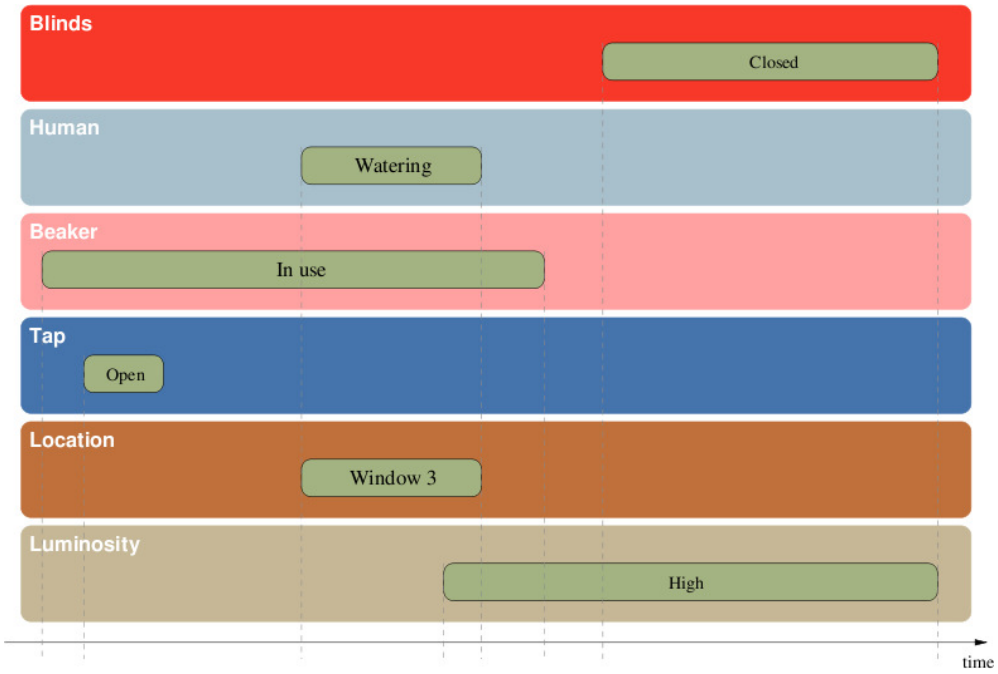

Figure 10. Sensor trace and inferred state and user's behavior (timelines) for the plant watering example.

When a sufficient amount of representative timelines and associated evaluations of the success of the synchronization (training data) is delivered to the Learning Layer, they can be used to learn a new weight for the evaluation. Each element in the dependencies will be used as an input of the ESN predicting the synchronization weight. Moreover, the domain provides information as to how these nodes are wired, whereby a dependency between nodes A and B in the Learning Layer exists iff a synchronization states that concept A depends (through some temporal constraint) on concept B.

For instance, such a dependency in the example above is the “watering” event, which can be associated to an output of the Learning Layer through appropriate training (in-factory or incremental), and “location”, which would likely be a learning module that is trained to recognize the location of the person from lower-level sensors. The accumulated timelines can then be provided to the Learning Layer to train the appropriate learning modules.

Although training of the Learning Layer is performed based on data obtained as a result of crisp inference, we expect to have a positive effect with respect to the ability in generalizing the predictions to slightly different sensory measurements as well as in terms of robustness to noise.

Indeed, the data-driven inference of the Learning Layer does provide an advantage, namely that evaluation of the situation is provided with a confidence value, whereas the crisp inference performed by the Control Layer will accept all and only those timelines that fall within the strict bounds of the Allen's Interval relations used to model the relations in the synchronizations. Conversely, the Learning Layer will most likely associate a lower confidence to the inference rather than discarding it completely.

4.5.5 Adaptation through learning

In summary, we propose to employ crisp (but highly intuitive) temporal models of how the system should behave for “bootstrapping” the Learning Layer.

Some of these models state the dependencies that a human domain expert has identified between sensors and actuation. Feedback from the execution of plans obtained through this “naive” model provides training data for the Learning Layer, which, over time, will refine the BDI agents' ability to choose the right plan.

The process will lead to a system which is driven by easily identifiable (albeit rough) rules specified in the form of temporal synchronizations, but which also delegates to data-driven inference the final choice of actuator plans.

Some of these models state the dependencies that a human domain expert has identified between sensors, and between sensors and context of the human being (e.g. activities being performed by the user(s)). In such a case, traces of sensors and inference over time provide an initial set of training data.

In both cases, after sufficient training data has been accumulated, new parts of the Learning Layer can be instantiated and trained with this information, in a process that delegates symbolic reasoning to data-driven inference for the purpose of increasing flexibility and adaptation.

While these are only first steps in making the RUBICON system more adaptive to real-world contexts, they opens up to several interesting possibilities, including the possibility to refine the learned model in the Learning Layer further through interaction with the Cognitive Layer and possibly new information modelled in the Control Layer.

4.6 Robotic Interaction

4.6.1 Overview of Agent BDI Hardware interaction

When the service layer has committed to the execution of a plan as given by the planner or plan database it will execute and monitor the individual actions and the collaborative data flows required in the given plan.

A plan will in this context consist of a number of subtasks that are executed in a partially ordered sequence (with temporal constraints giving the specific order of overlapping tasks). These individual subtasks fall into two categories:

- Actions and functionalities that are to be executed within the same host as the given agent.
- Actions and functionalities that are delegated to other agents participating in the ecology.

For the former, the service level BDI will utilize the PEIS-init components to start and stop local processes that execute the individual tasks and will monitor their outcomes and progress. For the later, this BDI will instead delegate the task to other agents in the ecology and rely on their explicit status update messages to be notified of failures and possibly progress in the execution. The monitoring of individual subtasks is performed in conjunction with the temporal reasoner that have an explicit model of the constraints of the tasks and that can model success and failures of the execution.

During the executions, the BDI agent will interact with the hardware only in terms of launching the appropriate processes or delegating tasks and by relying on their status updates which are

communicated through the tuplespace. These processes will in turn perform all low-level hardware access and will exchange all required data to perform computations on this data.

In order for these devices to collaborate to accomplish a given tasks, all such communication between the devices will be performed over the tuplespace and use the *meta-tuple* mechanisms to allow for dynamic reconfigurations of which devices should rely on what data. The specific configuration plan and deployment of these configurations are performed by the service level BDI.

For the case of tasks delegated to remote agents, these agents will provide the name of the output and input tuples which will accomplish the subtasks – and will in turn configure these tuples to be used as required by any of the internal processes they will run. This is describe in further detail in Section 4.9.3.

Although many robotic components, hardware accesses and algorithms have already been implemented for the PEIS middleware we see a need for a generic mechanism that allows for the reuse of the large range of software developed for other robotic middlewares such ROS and Player/Stage. For this purpose we provide a mechanism by which they can be reused within the RUBICON middleware preserving the advantages of their current implementations while gaining the advantages in terms of reconfigurability, fault recovery and scalability provided by the RUBICON middleware.

4.6.2 Utilizing ROS as hardware access provider

The Robot Operating System (ROS)[8] is a software framework to facilitate the creation of robot applications. ROS offers hardware abstraction, libraries, device drivers, message-passing, visualizers, package management and is continually being added to. ROS is released under the open source BSD license by Willow Garage and has become a de facto standard software tool in Robotics research, with applications starting to emerge within industry.

ROS was introduced to address a major issue in robotics, code reuse. This is a non-trivial problem due the varying of hardware from robot to robot. ROS is referred to as an operating system, but is not in the traditional sense. ROS provides a structured communication layer instead of process management and scheduling. ROS sits as a layer above the heterogeneous layer of operating systems and hardware implementations to provide transferable applications. The implementation is based on a graph architecture, within which each node of the graph presents a service. With message passing, the nodes then will be able to receive, post and multiplex sensor, control, planning, state and actuator messages. One of the many strong points of ROS is its hardware abstraction.

As such the use of ROS within the RUBICON project is of great interest. However, using the ROS middleware *as is* poses a problem in that it lacks support for many of the specific features of the PEIS middleware such as dynamic reconfiguration, multi-hop routing and discovery as components join and leave the ecology.

For this reason, we will develop a solution which allows for the re-use of ROS components for all low-level hardware and algorithm purposes, while retaining the advantages of the PEIS middleware. Our solution is based on a general bridging mechanism which allows for ROS components to

[8] http://www.ros.org

communicate with other ROS components via the PEIS middleware as well as to communicate and collaborate with dedicated PEIS components and with the WSN and WSAN motes available in the ecology. This will allow the service level BDI of to use the PEIS tuplespace to start, give parameters and to reconfigure the low-level ROS algorithms within the same agent as well as between different agents.

ROS implements the topic-based publish-subscribe model. This is considered to be a flexible means of communication, with a broadcast routing scheme. In ROS, the nodes with the publish-subscribe model are described as services. A service is described by a string name, and a pair of strictly typed messages. One of the messages is for requests and the other is for responses. The messages being passed between nodes, are a strictly typed data structures. The primitive types are supported in these structures, which includes integers, booleans and floating points. In addition to this, arrays of primitives and messages are also supported, allowing for nesting.

The planned implementation will have a layer which will handle the translations between the ROS services and the PEIS tuplespace. This layer will take the form of a ROS service itself. For the purposes of illustration, we will describe three ROS services on a mobile robot as ROS-1, ROS-2 and ROS-3. ROS-1 publishes odometry data from the wheels of the robot. ROS-2, controls the wheels of the robot. ROS-3, publishes data from a range sensor. This bridge layer service will be referred to as the ROS-Bridge and PEIS is referred to as PEIS-Tuplespace. The ROS-Bridge manages the subscriptions and publications of each service within the robot and thus provide a means of encapsulation for the robot.

With ROS-1, the ROS-Bridge subscribes to this service to start to get the *(x,y)* position of the robot. With ROS-2, it subscribes to the ROS-Bridge to get information on when to move in any particular direction. With ROS-2, this information is coming from somewhere in the PEIS tuplespace. With ROS-3, we have a similar situation as to ROS-1, it still subscribes to the ROS-Bridge, but the data volume would be large, possibly introducing latency.

Every time we have a ROS component, we translate the message coming from ROS-1 and ROS-3 and we create a tuple for each. This configuration means minimal alterations will have to be made to ROS-1 and ROS-3, which could very well already be natively implemented, harnessing code reuse.

With ROS-2, a subscription will be made to ROS-Bridge, which would be listening to the PEIS-Tuplespace for the corresponding tuples and, every time these are received, parse them and a simple parsing and translation will occur. A description for the expected tuple would be on ROS-Bridge, and a conversion to the service data structure would be defined and implemented. Initially, this will be a static implementation. Ideally, an automated conversation between tuples and message will be implemented, with future research work looking into the evolution of a dynamic translation.

When communicating over a network, or passing information between components, there is the possibility of latency. The effects of the latency could either be in regard to the internal implementation of either PEIS or ROS, but also control latency. Both PEIS and ROS are assumed to have been developed with their own internal latencies taken into account. It is the intention of our combined implementation, to have internal elements of one reliant on the internal elements of another. This has been taken under consideration, and will be investigated with care during implementation. Already, an implementation of a shared memory link layer has been suggested as a means to reduce latency on the PEIS side.

The control latency can also be described in relation to robot motion control. With a Humanoid Nao robot, coordination and fine timing are necessary to allow it to walk. This walking system could be implemented on the ROS side, and it has been debugged to do so effectively, move an element of that system remotely to PEIS and it could possibly no longer function. Within RUBICON, we believe that these rapid control scenarios are unlikely to occur to the level described with the Nao. Low level rapid movement would be maintained internally within a dedicated ROS unit, harnessing already existing ROS implementations. *This provides a trade-off between the amount of reconfigurability that can be introduced into existing ROS components and the effective latencies.*

4.6.3 Intra-agent interaction

In order to allow intra-agent collaborations, as opposed to collaboration between processes on the same host, the service level BDI and planner allows for the delegation of tasks to other agents. In terms of the planner this is done by using introspection to detect the domain (semantic) description of the services and functionalities exported by all agents in the ecology and the associated *tuple names* that list the functionality output and status updates of these agents.

In the execution of plans that involve tasks executed by other agents, a service level BDI agent will read the corresponding *tuple names* as provided by the remote agent and write these into the configurations of the locally running processes that require these connections.

When an agent is executing a task given by a coordinator BDI agent or a task given by another service level agent it will start any local processes (possibly also invoke further remote agents) for the execution. In the plan computed to accomplish this task it will have the output tuples of the corresponding tuples and will list the names of these tuples it it's tuplespace for the calling processes to read into their configurations.

***Example**: Assume that the robot A is given the goal to be present in the kitchen. To accomplish this goal, the robot will need a navigation module that needs obstacle information as well as a source of localization information. Since it lacks localization functionalities it will invoke the use of another agent, agent B, that can export localization functionalities for the robot in the given target area. This second agent can be another robot with a camera, a statically mounted movement sensor or any other type of agent.*

Next, agent A will create a configuration in which the navigation component obstacle input information is connected to the output tuples of a local sonar process. Furthermore, it will query agent B for the name of the corresponding localization output tuples, and will input this name into the navigation modules localization input.

Finally, it will monitor the status tuples of the navigation component, the sonar component and the status tuple of agent B. If either of these status messages indicate a failure it will trigger a local re-planning.

4.6.4 Interaction with WSN motes

Due to their simplicity and the difficulties in fitting formal semantic descriptions and data languages into the computationally constrained WSN devices a combination of local autonomy and delegation of responsibilities will be used. For this purpose these devices can be seen as distributed agents,

where certain local autonomy decision is computed onboard and certain decisions are delegated to the an external so call *proxy process* that will run onboard PC devices in the ecology.

The details of the discovery of a need for proxy devices, the launching of this process and the communications between PC devices and proxy and proxy and proxied device is described in detail in deliverable D1.1.

From a birds eye perspective a WSN mote in RUBICON will contain the 802.15.4 based communication of the communication layer (WP1), the Learning Layer (WP2) as well as a basic version of the service level BDI from the Control Layer (WP3). Furthermore, depending on the local application and hardware of the devices they will contain software for sensing and/or actuating with this hardware and will accept specific MaD/WiSE stream system messages that are implementation dependant for the different purposes of the different types of motes.

Since these devices lack direct access to the distributed tuplespace some of the aspects of the communication with this service level BDI as well as tuple based communication with the basic sensing and actuation components of them is delegated to the external proxy process. These external proxy processes will perform pure translation tasks between tuple representations and the implementation dependent (nesC) messages for the agents. Furthermore, they will publish the semantic descriptions of the WSN agents and contain all higher level planning activities for the agents.

4.7 Reference Control Architecture

4.7.1 Service and coordination agents for non-computationally-constrained devices

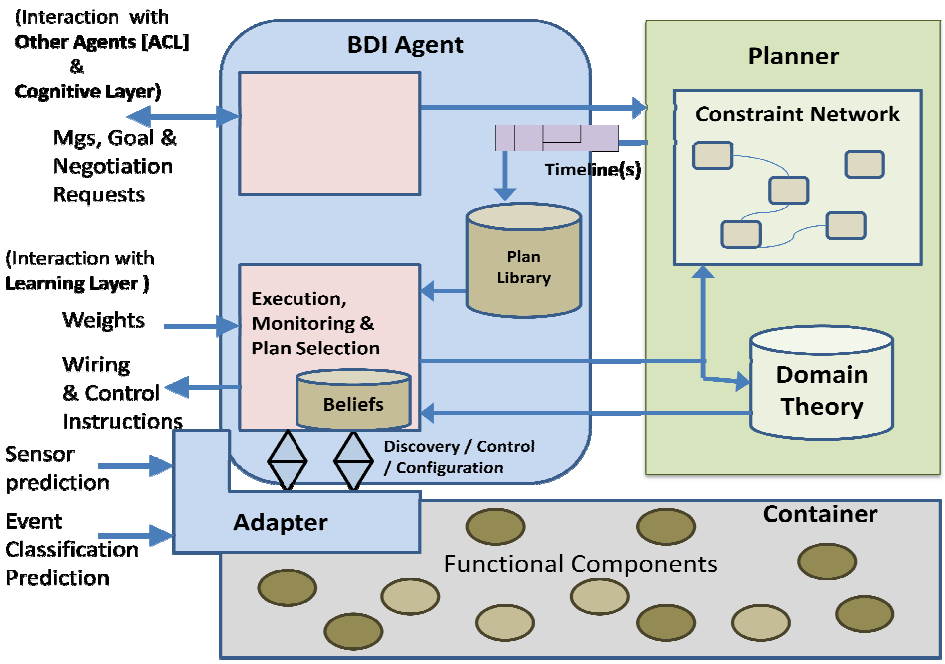

Figure 11. High-Level Agent Control Architecture

In order to summarise some of the discussions carried out in this section and allow us to identify a number of issues we need to tackle in the remaining of this project in order to integrate our BDI and planning solutions, Figure 11 illustrates the high-level, agent control architecture to be for used for both coordinator and service agents, at least those to be deployed on non-computationally constrained devices and robots.

The BDI agent is the front-end interface toward (i) other agents, and (ii) (at least for the coordinator agent) toward the Cognitive Layer. Coordination and service agents exchange messages, conveying requests to achieve goals, and/or used to negotiate and synchronize task execution and collaborations across multiple service agents. Each BDI agent keeps a queue of incoming goal requests, while the goals it has actually committed to achieve (**agent's commitments**) are stored into the agent's **commitment memory**.

The BDI agent is also responsible to **interface** with the set of **functional components** controlled by the agent control system (sensor, actuators, but also software components delivering perception capabilities, such as localization and object recognition, and sensory-motor behaviours used for safe navigation, manipulation, etc), and also with the Learning Layer's interfaces providing sensor and classification event predictions. In order to extend the RUBICON's control over heterogeneous (robotic and not) functionalities, rather than interfacing these functionalities directly with the BDI agent architecture, these interfaces are fitted into a RUBICON-specific **adapter component**. Such an approach has the advantage that standard means (such as the PEIS-init discussed in Section 3.1.5, the EIS environment interface [48] , or the Streams system described in D1.1 for computational constrained devices) can be used to interact with the underlying functionalities. The common assumption will be that each adapter will export a **container**-style API to (i) retrieve descriptions of all the available functional components (both *active/ready to execute*, or *inactive*, which need to be loaded from the file system) together with their semantic descriptions, (ii) control the life-cycle of these components (*load/unload/activate/de-activate*) and configure them (by updating functional and wiring parameters, such as meta-tuples used to define PEIS configurations). An existing methodology that NUID UCD has already applied in a number of application to interact with functional components operating within a component based framework [83] will be adapted to operate with the PEIS-Init discussed in Section 3.1.5), as well as with mainstream component frameworks, such as those promoted by the Open Service Gateway initiative (OSGi)[9], hosting 3rd-party data analysis functionalities, such as those used for sound recognition developed by TECNALIA.

A **plan library** will be used to cache a number of feasible temporal plans (timelines) at any given time. These plans will include both application plans to pursue **application goals**, and **internal plans**, which carry out a number of **internal processes**. These processes subtend the inner working of the agent control system by operating on the agent's internal state - a common feature in BDI architectures extending the classic approach of PRS-like systems, such as NUID UCD's AF.

Within the RUBICON's control architecture, this will be used to define a number of internal processes:

Commitment Reconsideration Process - The reception of each new goal or task request should trigger a reasoning process to decide which goals to commit to and which goal to suspend, or indeed drop, based on the available resources, and the priorities and temporal information associated to

[9] http://www.osgi.org/

each goal. For instance, the coordinator agent may decide to drop a low-priority goal whenever a more urgent one is received and it does not have enough resources (i.e. service agents) to pursue both goals. It may also have to choose a sub-set of goals to pursue out of a set of goals with the same priority. BDI agents will use the planner to find out if a goal can be scheduled and performed together with the ones already considered by the RUBICON ecology. To do this, each goal needs to be translated in a suitable form and posted to the planner.

The coordinator agent may need to evaluate different combinations of goals, in order to find the combination that will maximize the number of high-priority goals actually pursued with the resources available to the RUBICON. In addition, the planner will typically find multiple options to satisfy each request. For this reason, the BDI-agent/Planner controller will need to evaluate different options in terms of resource utilization, and the global time span of the corresponding timeline. Crucially also, the learning layer should be exploited to estimate the costs and/or the probabilities of success in each of the step considered by each plan. For this to happen, we envisage that the planner will use information that will be extracted from the Learning Layer and cached in order to be used as planning time. In addition, each candidate timeline found by the planner may be checked against the actual output of the Learning Layer, i.e. to identify the timeline that most closely fits with the output of the Learning Layer.

Introspection Process. When asked, e.g. by the co-ordinator, if it would be able to carry out a given goal given its current situation, each service agent will use its own planner to check if it can find a suitable plan to satisfy the goal. By leveraging the Learning Layer, the service agent should find give an estimate of the probability of success, if it were to pursue the goal.

Monitoring Process. This process will monitor for significant events, such as the robot reaching given way-points or entering in wireless radio contact with a new sensor, drop of performances, as well as the achievement of goals and other important application-specific events. It will then notify these events to the coordinator agent (for service agents) or to the Cognitive Layer (for coordinator agents).

4.7.2 Service agents for computationally constrained devices

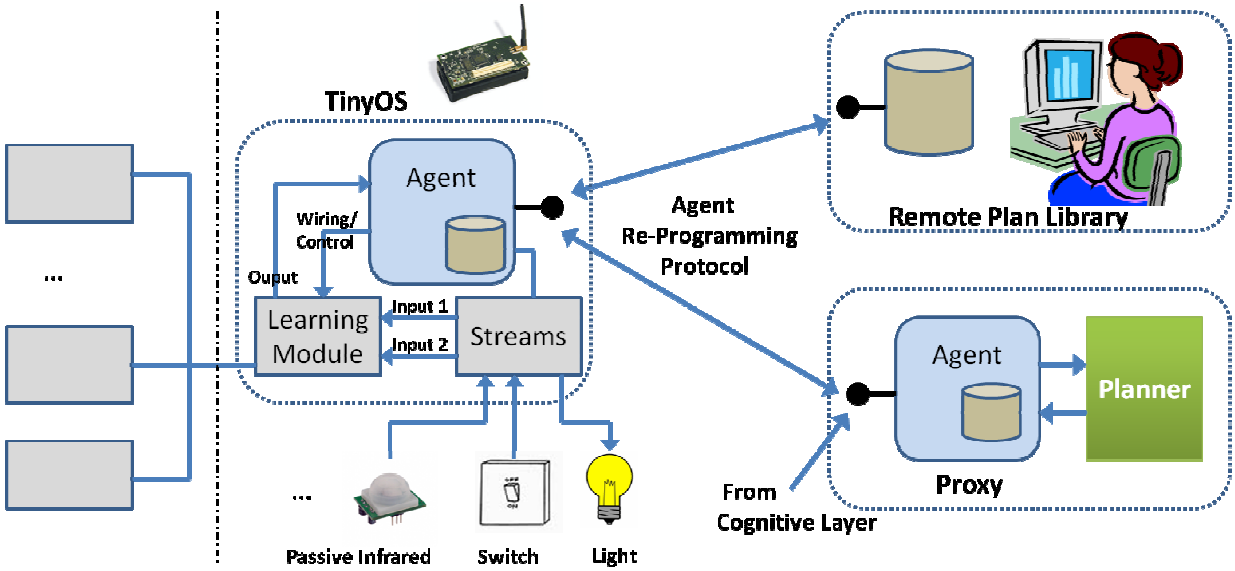

Figure 12. Reference Control Architecture for Computationally Constrained Devices.

Computationally constrained devices are already seamlessly integrated within the Control Layer by using the proxy facility described in Deliverable D1.1.1. However, at least parts of the Control Layer will need to be directly deployed on these nodes, in order to more closely interact with the instances of the Learning Layer installed on them, and also to provide autonomous and fast action decision loops without requiring constant interaction with their proxy interface.

For these reasons we envisage the use of micro-BDI solutions (as depicted in Figure 12), to target WSN nodes running the TinyOS platform, as well as Java-enabled nodes, based on the Sun SPOT specifications[10].

As a way of example, the pro-active control of lights exhibited in the LIGHTS case study may be tackled by deploying one of such micro-BDI solution to: (i) monitor a passive infrared sensor triggered by the presence of the user, and (ii) the use of the light switch. These sensors will be interfaced via the Streams system (described in Deliverable D1.1), and routed to one of the local learning modules (described in Deliverable 2.1) with a single output. The learning module will be also connected to learning modules located in neighbouring nodes, e.g. adjacent rooms, and installed on similar, light-controlling nodes.

In general, the micro-BDI agent logic will need to supervise:

- the creation of learning modules and their wiring to the local sensors
- the creation of the outputs to be employed in the specific control application

[10] http://www.sunspotworld.com/

- the training of such outputs, as specified by application-specific plans.

The deployment of such micro-BDI solutions can be supported in two ways:

1. In the first instance, these solutions can be pre-programmed and installed on the WSN node via agent re-programming protocols.

2. In the second instance, these solutions can avail of proxy service agents, with full planning capabilities (which can also interact with the RUBICON's Cognitive Layer) to find plans at runtime, and install them on selected WSN nodes of the RUBICON.

5. Test Plan

This section introduces an initial set of integration test specifications designed as part of Task 3.1, and which includes test specifications for the integrated RUBICON middleware delivered by WP3.

The design of this test plan has been guided by the following requirements:

- **modular** - the test plan should be easily extendible, to accommodate new test cases. Tests' definitions should also re-use the results of previous test, in order to build a test suite that would test all RUBICON's features in a modular and optimized manner.
- **automated** - tests should be as automated as possible, requiring none or minimum human supervision, in order to be used for quick regression testing and performance assessment.

In particular, the goal of the **middleware test specifications** presented in this deliverable is to help us to uncover errors associated with the RUBICON's communication capabilities developed in WP1, and the interfaces we have defined among the RUBICON's architectural layers, before each WP will define their own unit testing to test the working of the components inside each layer. The emphasis in the middleware tests documented here is to test if the communication capabilities can support the defined inter-layer.

5.1 Integration Tests

Integration test cases were chosen by examining the interface requirements described in the deliverable D2/3/4.1, and the capabilities of the Communication Layer described in D1.1, but do not consider communication functionalities already covered by pre-existing components (such as PEIS).

Rather than through the execution of complete case studies involving all the layers in the RUBICON's architecture, these tests will be supported by dedicated test programs purposely written to exercise the middleware and layers' APIs via selected inputs.

Each test contains the following information:

- **Test Case ID:** The unique id of the test.
- **Focus:** A description of the area of focus of the test, for instance, if the test focuses on the verifying the interface between the Control and the Communication layer.
- **Type of test:** A test can be either *positive* or *negative*, or both. Positive test cases usually test a requirement from within normal circumstances. Negative tests usually test how the system reacts to anomalous input or to error conditions.
- **Purpose**: A description of the specific purpose of the test.
- **Testing environment:** Description of the environment in which the test case should be run. It contains information of how many and what kind of nodes (WSAN motes, PCs) and what software (including both the RUBICON middleware and test programs) must be installed on them.
- **Initialization:** Description of the initial conditions the system and/or the middleware must be in before the test can begin and after all the required software has been installed on all the nodes involved in the test.
- **Input (optional)**: Specifies any inputs/actions required to initiate the test.
- **Process:** Each test consists of a test process that can be specified in one or more steps performed by the test programs. This section contains the pseudo-code of all the test programs introduced in

each test. Each test program is also assumed to report any exception to the control flow described in its pseudo-code.

- **Expected result,** summarising the correct behaviour of the system after the task has been executed.

Test Case ID:	TDiscoveryMote	**Focus**	Component Mgmt. - Integration with WSAN	**Type**	Positive
Purpose	Test the ability to discover a mote newly connected to the WSAN				
Testing Environment	Mote 1, base station connected to a PC via USB and interfaced with a number of sensors and actuators. Mote 2, connected to a powered USB hub. Software Mote 1: Communication Layer and test program (PTDiscoveryMote.WSAN). Software Mote 2: Communication Layer Software PC : TinyOS SDK and test program (PTDiscovery.PC).				
Initialization	Run test program PTDiscoveryX.PC on the PC side.				
Input	Power USB hub.				
Test Process	**PTDiscoveryMote.WSAN**: 1) listen for mote discovery events (D1.1 /5.3.4/*moteJoined*) and forward them to the USB port **PTDiscovery.PC**: 1) open USB connection and repeat until exit signal 2) read data from USB port				
Test Results	Test program PTDiscovery.PC will receive (within few seconds from the powering of the USB hub) the description of the newly connected mote, including the list of all the sensors and actuators interfaced with the mote				

Test Case ID:	TSensorX	**Focus**	Streams - Integration with WSAN transducers	**Type**	Positive
Purpose	Test reading raw sensor data from sensor *X* . This test case specifies a family of tests to be repeated for each of the sensors interfaced with WSAN nodes (*X* ∈ {temperature, humidity, light, motion ...})				
Testing	Single mote connected with a PC via USB and interfaced with the sensor under test.				

Environment	Software Mote : Communication Layer and test program (PTSensorX.WSAN). Software PC : TinyOS SDK and test program (PTSensorX.PC).
Initialization	Run test program PTSensorX.PC on the PC side.

Test Process	**PTSensorX.WSAN**:
	1) open (D1.1/5.3.2/*openLocal*) a stream connection to read from sensor *X*
	
	2) repeat N times
	{
	2.1) read the data from the stream to sample sensor *X*
	2.2) write data to USB together with the timestamp of the time the data was acquired by the sensor
	}
	3) close stream and signal exit
	
	PTSensorX.PC:
	1) open USB connection and repeat until exit signal
	{
	1.1) read data from USB, and measure delays between readings
	}
	2) Compare delays with reference delays

Test Results	Test program PTSensorX.PC will receive N data updates. Delay between data timestamp and the time the data is accessed by the test program will be within given tolerance (specific for each sensor).

Test Case ID:	TSensorXError	**Focus**	Streams - Integration with WSAN transducers	**Type**	Negative
Purpose	Test reading raw data locally from a specific sensor that is not interfaced with the mote This test case specifies a family of tests to be repeated for each of the sensors interfaced with WSAN nodes (e.g. temperature, humidity, light, motion ...)				
Testing Environment	Same as TSensorX (only Mote side)				
Initialization	Same as TSensorX				
Test Process	Same as TSensorX				
Test Results	Exception when opening stream				

Test Case ID:	TActuatorX	**Focus**	Streams - Integration with WSAN	**Type**	Positive

			actuators		
Purpose	Test writing control data to actuator *X* installed on a local WSAN node. This test case specifies a family of tests to be repeated for each of the actuator types interfaced with WSAN nodes (*X* ∈ { switch, dimmers,...})				
Testing Environment	Single mote connected with a PC via USB and interfaced with the actuator under test. Software Mote : Communication Layer and test program (TActuatorX.WSAN). Software PC: TinyOS SDK and test program (TActuatorX.PC).				
Initialization	Run test program PTActuatorX.PC on the PC side.				

Test Process	**PTActuatorX.PC**:
	1) Repeat N times or until PTActuatorX.WSAN signals error
	{
	1.1) write control data to USB port
	1.2) read data from USB port
	1.3) signal error if control data is different from data read
	}
	
	PTActuatorX.WSAN:
	1) open (D1.1/5.3.2/*openLocal*) a stream connection to the actuator *X*
	2) Repeat
	{
	2.1) read data from USB and write it to the actuator stream
	2.2) read actuator status and forward it to USB
	}
	3) close stream				

Test Results	Test program PTActuatorX.PC will perform N steps without exceptions.				

Test Case ID:	TRemoteSingleIslandStream	**Focus**	Streams - Integration with WSAN	**Type**	Positive
Purpose	Test communicating with a remote WSAN node connected on the local island via unreliable stream communication (PC-to-mote base station-to-mote)				
Testing Environment	Mote 1, base station mote connected with a PC via USB. Mote 2, within radio range of the base station. Mote 1: Communication Layer and test program (PTRemoteSingleIslandStream1.WSAN). Mote 2: Communication Layer and test program (PTRemoteSingleIslandStream2.WSAN).				

	PC: TinyOS SDK and test program (PTRemoteSingleIslandStream.PC).
Initialization	Run test program PTRemoteSingleIslandStream.PC on the PC side.

Test Process	**PTRemoteSingleIslandStream.PC**:
	1) Repeat i=1..N
	{
	Repeat // keeps trying sending data and waiting for feedback
	{
	1.1) write i to USB port
	1.2) read data from USB port, or timeout after 2 second
	} until read data == i
	}
	
	PTRemoteSingleIslandStream1.WSAN:
	1) open (D1.1/5.3.2/*open_remote*) a remote stream connection toward Mote 2
	2) Repeat
	{
	2.1) read data from USB and write it to stream
	2.2) read data from stream and forward it to USB
	}
	3) close stream
	
	PTRemoteSingleIslandStream2.WSAN:
	1) open (D1.1/5.3.2/ *open_remote*) a remote stream connection toward Mote 1
	2) Repeat
	{
	2.1) read data from stream
	2.2) forward data to stream
	}
	3) close stream

Test Results	**PTRemoteSingleIslandStream.PC** succesfully completes N steps

Test Case ID:	TRemoteSingleIslandMsg	**Focus**	Msgs - Integration with WSAN	**Type**	Positive
Purpose	Test communicating with a remote WSAN node connected on the local island via reliable (connectionless) stream communication (PC-to-mote base station-to-mote)				
Testing Environment	Mote 1, base station mote connected with a PC via USB. Mote 2, within radio range of the base station. Mote 1: Communication Layer and test program (PTRemoteSingleIslandMsg1.WSAN). Mote 2: Communication Layer and test program (PTRemoteSingleIslandMsg2.WSAN).				

	PC: TinyOS SDK and test program (PTRemoteSingleIslandMsg.PC).
Initialization	Run test program PTRemoteSingleIslandMsg.PC on the PC side.

Test Process	**PTRemoteSingleIslandMsg.PC**:
	1) Repeat i=1..N
	{
	1.1) write i to USB port
	1.2) read data from USB port
	}
	
	PTRemoteSingleIslandMsg1.WSAN:
	1) Repeat
	{
	1.1) read data from USB and forward it (D1.1/5.3.2/*send*) to stream
	1.2) upon reception of data from stream (D1.1/5.3.2/*receive*), forward it to USB
	}
	
	PTRemoteSingleIslandMsg2.WSAN:
	
	1) upon reception (D1.1/5.3.2/*receive*) of a message from Mote 1, send message back to Mote 1 (D1.1/5.3.2/*send*)

Test Results	**PTRemoteSingleIslandStream.PC** succesfully completes N steps

5.2 Test of Mockup Communication Layer

Figure 13 depicts the environment where we tested the mockup Communication Layer described in D1.1. For the experiment, conducted in NUID UCD, we used a TurtleBot robot based on the iRobot CREATE robot platform. We used ROS to tele-operate the robot to perform the path highlighted in Figure 13 after installing 8 WSN motes (telosB clones) running the mockup Communication Layer (the blue dots in the diagram). A base station WSN mote was also connected via USB to the netbook controlling the robot. The robot was instructed to stop at specific landmarks (the red dots in the diagram) for about three seconds, and use the mockup Communication Layer to query all the sensor data (temperature, humidity, luminosity, radio signal strength index) gathered by the nodes.

Round trip times between the issue of the request and the reception of the data varied between 10 and 954 milliseconds during the duration of the experiment.

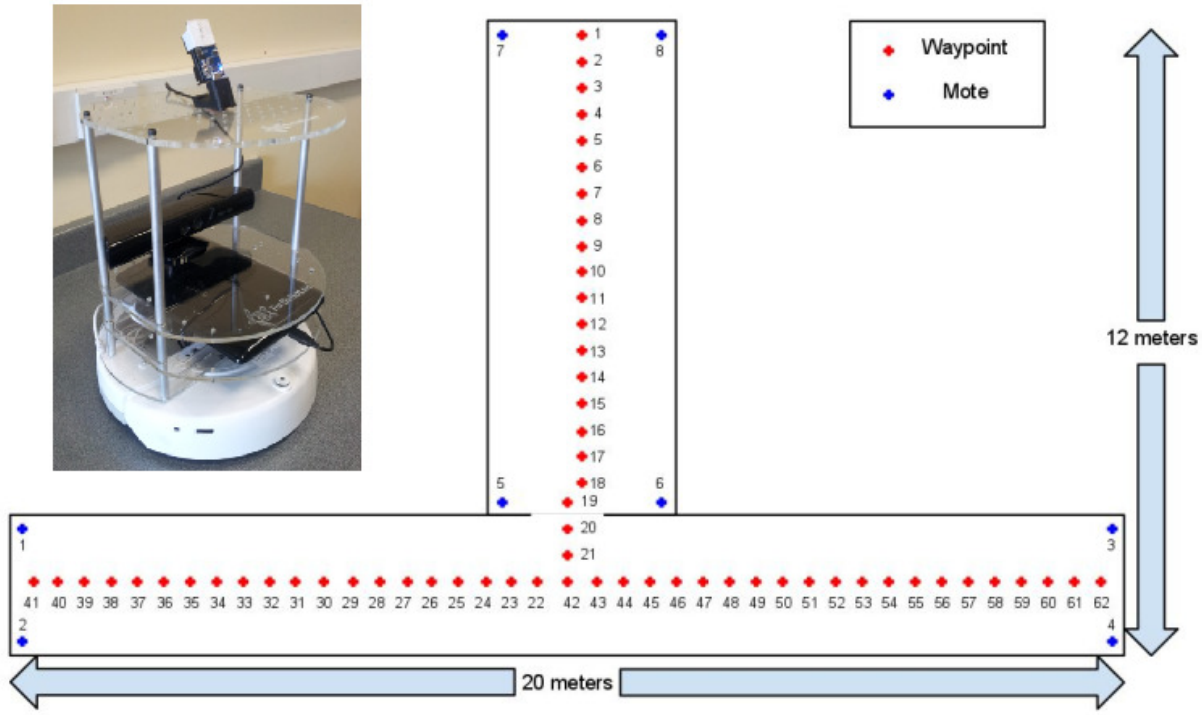

Figure 11. Layout of the experiment used to test the Mockup Communication Layer, and the TurtleBot robot used in the experiment (in upper left corner).

6. References

[1] Bordini, R. H., Dastani, M., Dix, J. and Amal El Fallah Seghrouchni. (2005) eds. *Multi Agent Programming: Languages, Platforms and Applications*. Springer-Verlag, 2005.

[2] H. M. Adnan, "A Planning Component for CLAIM Agents," in Proceedings of International Workshop On Multi-Agent Systems Technology and Semantics. IEEE Romania, 2009.

[3] Bratman, M.E. (1987) *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, USA, 1987.

[4] Bratmann, M.E., Israel, D.J., Pollack. (1988) M.E. Plans and resource-bounded practical reasoning. *Computational Intelligence, Vol.4, P349-355, 1988*.

[5] Busetta, P., Howden, N., R¨onnquist, R. and Hodgson. A. (1999) Structuring bdi agents in functional clusters. In *Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages - ATAL 99*, 1999.

[6] Castelfranchi, C.; Dignum, F.; Jonker, C. M.; and Treur, J. (1999) Deliberate normative agents: Principles and architecture. *In Proceedings of ATAL-99*. Weiss, G. 1999.

[7] Collier, R.W., O'Hare G.M.P., Lowen, T., Rooney, C.F.B., Beyond Prototyping in the Factory of the Agents, 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic, 2003

[8] Dastani, M., F. de Boer, F. Dignum, and J.-J. Meyer. (2003) Programming agent deliberation: An approach illustrated using the 3APL language. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS' 03)*, pages 97–104. ACM, 2003.

[9] Dastani, M., F. de Boer, F. Dignum, W. van der Hoek, M. Kroese and J.-J.Ch. Meyer. (2002) Programming the Deliberation Cycle of Cognitive Robots, *CognitiveRoboticsWorkshop*, 2002.

[10] De Silva, L., Padgham, L.: A comparison of BDI based real-time reasoning and HTN based planning. In: AI 2004: Advances in Artificial Intelligence, 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, Springer (2004) 1167-1173

[11] De Silva, L., Padgham, L.: Planning on demand in BDI systems. In: International Conference on Automated Planning and Scheduling, Monterey, California (2005)

[12] Dennett, D. C. (1987) *The Intentional Stance*. The MIT Press, Massachusetts, 1987.

[13] Despouys, O., and Ingrand, F. F. 1999. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *European Conference on Planning (ECP)*, 278–293.

[14] F. R. Meneguzzi, A. F. Zorzo, and M. D. C. M´ora. Propositional planning in BDI agents. In *Proc. of SAC'04*, p 58-63. ACM Press, 2004.

[15] Ferguson, Innes A. (1992) *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. Ph.D. diss., Computer Laboratory, University of Cambridge, Cambridge UK.

[16] Fikes, R. E. and Nilsson, N. J. (1971), STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208, 1971.

[17] Gallien, M.; Ingrand, F.; and Lemai, S. 2005. Robot Actions Planning And Execution Control For Autonomous Exploration Rovers. In *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, volume 603 of *ESA Special Publication*.

[18] Georgeff M.P. and Ingrand, F.F. (1989) Decision-making in an embedded reasoning system. *In Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI'89), pages 202–206, Detroit, Michigan, USA, 1989.

[19] Georgeff, M. P. and Lansky, Amy L. (1987) Reactive reasoning and planning: an experiment with a mobile robot. *In Proceedings of the 1987 National Conference on Artificial Intelligence* (AAAI 87), pages 677–682, Seattle, Washington, July 1987.

[20] Ingrand, F., Despouys, O.: Extending procedural reasoning toward robot actions planning. In: Proceedings of the 2001 IEEE International Conference on Robotics and Automation, Seoul, Korea (2001) 9–10

[21] Jensen, R., & Veloso, M. (1998) Interleaving deliberative and reactive planning in dynamic multi agent domains. *In Proceedings of the AAAI Fall Symposium on on Integrated Planning for Autonomous Agent Architectures*.AAAI Press, 1998.

[22] Lemai, S., and Ingrand, F. 2004. Interleaving temporal planning and execution in robotics domains. In AAAI.

[23] Kinny, D., Georgeff, M., Rao, A. (1996) A methodology and modeling technique for systems of BDI agents. *Proc. Of 7th European Workshop on Modelling Autonomous Agents in Multi-Agent Worlds* (MAAMAW'96), LNAI1038, Springer-Verlag, p56-71, 1996

[24] Móra, M. C. et al. (1998) BDI models and systems: reducing the gap. *ATAL'98*.

[25] Muldoon, C., O Hare, G.M.P., Collier, R.W., O Grady, M.J., Agent Factory Micro Edition: A Framework for Ambient Applications. In Intelligent Agents in Computing Systems, volume 3993 of Lecture Notes in Computer Science, pages 727–734, Reading, UK, 28-31 May 2006. Springer.

[26] Olivier Despouys , François Felix Ingrand, Propice-Plan: Toward a Unified Framework for Planning and Execution, Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning, p.278-293, September 08-10, 1999

[27] Paris.Pokahr, A., Braubach, L., Lamersdorf, W. (2005) A Goal Deliberation Strategy for BDI Agent Systems, *In Third German conference on Multi-Agent System TEchnologieS* (MATES-2005); Springer-Verlag, Berlin Heidelberg New York, pp. 82-94.

[28] Padgham, L. and Lambrix, P. (2000) Agent Capabilities: Extending BDI Theory in *Proceedings of Seventeenth National Conference on Artificial Intelligence - AAAI 2000*, Aug 2000, p 68-73

[29] Pokahr, A., Braubach, L., Lamersdorf, W. (2005) A Goal Deliberation Strategy for BDI Agent Systems, *In Third German conference on Multi-Agent System TEchnologieS* (MATES-2005); Springer-Verlag, Berlin Heidelberg New York, pp. 82-94.

[30] Rao, A. and Georgeff, M. (1992) An abstract architecture for rational agents. In Principles of Knowledge Representation and Reasoning: *Proceedings of the Third International Conference* KR 92, pages 439–449.

[31] Rao, A. S. (1996) AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of Modelling Autonomous Agents in a Multi-Agent World*, number 1038 in LNAI, pages 42–55. Springer Verlag.

[32] Rao, A. S. and Georgeff, M. P. (1991) Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, April 1991.

[33] Rao, A. S. and Georgeff, M. P. (1995) BDI agents: from theory to practice. In Victor Lesser, editor, In: *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), pages 312–319, San Francisco, CA, USA*, 1995. The MIT Press: Cambridge, MA, USA.

[34] Ross, R. J. (2003) *MARC -- Applying Multi-Agent Systems to Service Robot Control*. MSc. thesis, Department of Computer Science, University College Dublin.

[35] Saffiotti, A., Ruspini, E., and Konolige, K. (1993) Blending reactivity and goal-directedness in a fuzzy controlled*. In Proceedings of the IEEE Int. Conf. On Fuzzy Systems*, pages 134-139, CA, 1993. IEEE Press.

[36] Saffiotti, Konolige, K., Ruspini, E.H. (1995) A multivalued-logic approach to integrating planning and control. *Artificial Intelligence*, 76(1-2):481–526, 1995.

[37] S. Sardina and L. Padgham, "Hierarchical planning in BDI agent programming languages: A formal approach," in Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems. ACM New York, NY, USA, 2006, pp. 1001–1008.
[38] Shoham, Y. Agent oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
[39] Shoppers, M. (1987) Universal plans for reactive robots in unpredictable environments. *Proc. of IJCAI-87,* 1987.
[40] Thangarajah, J., Padgham, L., Harland, J. (2002) Representation and Reasoning for Goals in BDI Agents. *ACSC 2002*: 259-265.
[41] Andrzej Walczak, Lars Braubach, Alexander Pokahr, Winfried Lamersdorf, Augmenting BDI Agents with Deliberative Planning Techniques, in The 5th International Workshop on Programming Multiagent Systems (PROMAS), 2006.
[42] Wilkins, D. E. (1985) *Hierarchical Planning: Definition and Implementation*, Number 370, December 1985.
[43] S. Sardina and L. Padgham, "Hierarchical planning in BDI agent programming languages: A formal approach," in Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems. ACM New York, NY, USA, 2006, pp. 1001–1008.
[44] H. M. Adnan, "A Planning Component for CLAIM Agents," in the Proceedings of International Workshop On Multi-Agent Systems Technology and Semantics. IEEE Romania, 2009.
[45] Bellifemine, F., Poggi, A. and Rimassa, G. (1999) JADE - A FIPA compliant agent framework. *Proc. of 4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, 1999.
[46] Poslad, S., Buckle, P. and Hadingham, R. (2000) *The FIPA-OS agent platform: Open Source for Open Standards*, published at PAAM2000, Machestor, UK, April 2000.
[47] Ricci, A., Piunti, M., Acay, L. D., Bordini, R. H., Hübner, J. F., Dastani, M., Integrating heterogeneous agent programming platforms within artifact-based environments. AAMAS (1) 2008: 225-232.
[48] Behrens, T., Hindriks, K., Dix, J., Dastani, M., Bordini, R., Hübner, J., Braubach, L., Pokahr, A.: An interface for agent-environment interaction. In: Proceedings of the The Eighth International Workshop on Programming Multi-Agent Systems (2010).
[49] Nilsson, N. J. (1994) Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research*, 1, pp. 139-158, January 1994.
[50] Rao, A. S. (1996) AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of Modelling Autonomous Agents in a Multi-Agent World*, number 1038 in LNAI, pages 42–55. Springer Verlag
[51] Felipe Meneguzzi, Michael Luck, Leveraging new plans in AgentSpeak(PL), in proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT) (2008).
[52] F. Pecora and M. Cirillo. A Constraint-Based Approach for Plan Management in Intelligent Environments. In Proc. of the Scheduling and Planning Applications Workshop at ICAPS09, 2009.
[53] Muscettola, N.; Dorais, G. A.; Fry, C.; Levinson, R.; Plaunt, C.: IDEA: Planning at the Core of Autonomous Reactive Agents. In: 3rd International NASA Workshop on Planning and Scheduling for Space. 2002.
[54] Fratini, S.; Pecora, F.; Cesta, A.: Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. In: Archives of Control Sciences, 18(2), 2008, 231-271.
[55] Jónsson, A., Morris, P., Muscettola, N., Rajan, K., and Smith, B. (2000). Planning in interplanetary space: Theory and practice. In Int. Conf. on Artif. Intell. Planning and Scheduling, pages 177–186.
[56] Bresina, J. L., Jónsson, A. K., Morris, P. H., and Rajan, K. (2005). Activity planning for the mars exploration rovers. In Biundo, S., Myers, K., and Rajan, K., editors, ICAPS, pages 40–49. AAAI.

[57] Williams, B., Ingham, M., Chung, S., and Elliott, P. (2003). Model-based programming of intelligent embedded systems and robotic space explorers. In Proc. of IEEE Special Issue on Modeling and Design of Embedded Software, pages 212–237.
[58] Rabenau, E., Donati, A., Denis, M., Policella, N., Cesta, A., Cortellessa, G., Oddi, A., Fratini, S., and Bernardi, G. (2009). The raxem tool on mars express - uplink planning optimisation and scheduling using ai constraint resolution. In Proc. of the 6th Int'l Workshop on Planning and Scheduling for Space, IWPSS-09.
[59] McGann, C., Py, F., Rajan, K., Ryan, J., and Henthorn, R. (2008). Adaptive control for autonomous underwater vehicles. In Proc. of the 23rd Nat. Conf. on Artif. Intell. (AAAI), pages 1319–1324.
[60] Doherty, P., Kvarnström, J., and Heintz, F. (2010). A temproal logic-based planning and execution monitoring framework for unmanned aircraft systems. Journal of Automated Agents and Multi-Agent Systems, 2(2).
[61] Dechter, R. (2003). Constraint Processing. Morgan Kaufmann.
[62] Alan H. Bond and Les Gasser. Readings in Distributed Artificial Intelligence. Morgan Kaufmann Publishers, San Mateo, CA, USA, August 1988.
[63] Michael Wooldridge. An Introduction to MultiAgent Systems. John Wiley & Sons, New York, NY, USA, June 2002.
[64] Gilad Zlotkin and Jeffrey S. Rosenschein. Negotiation and task sharing among autonomous agents in cooperative domains. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, pages 912–917, San Mateo, CA, 1989. ACM.
[65] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers, C-29(12):1104–1113, 1981.
[66] Cohen P. R. and Levesque. H. J. (1995) Communicative actions for artificial agents. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems* (ICMAS'95), pages 65–72, San Francisco, CA, USA, 1995. The MIT Press: Cambridge, MA, USA.
[67] Castelfranchi, C. (1995) Commitments: From individual intentions to groups and organizations. *In Proceedings of the First International Conference on MultiAgent Systems (ICMAS-95)*, pages 41-48, Menlo Park, California, June 1995.AAAI Press.
[68] Haddadi, A. (1996) Communication and Cooperation in Agent Systems: A Pragmatic Theory. Number 1056 in *Lecture Notes in Computer Science*. Springer-Verlag: Heidelberg, Germany, 1996.
[69] Levesque, H., Cohen, P., Nunes J. (1990) On Acting Together. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pp. 94--99, Boston, MA, 1990.
[70] J. Rashid, M. Broxvall. Indirect Reference: Reconfiguring Distributed Sensors and Actuators. In Proceedings of the Third IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC2010), California, USA, June 2010.
[71] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B.S. Seo, Y.J. Cho. The PEIS-Ecology Project: Visions and Results. In Proceedings of The 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2008), pp. 2329-2335, Nice, France, September 2008.
[72] R. Lundh, L. Karlsson and A. Saffiotti. Plan-Based Configuration of an Ecology of Robots. Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA) pp. 64-70. Rome, Italy, 2007.
[73] L. Busoniu, R. Babuska, B. De Schutter, A Comprehensive Survey of Multi-Agent Reinforcement Learning. IEEE
[74] Transactions on Systems, Man, and Cybernetics — Part C: Applications and Reviews, vol. 38, no. 2, pages 156–172, 2008
[75] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In Proceedings of the Tenth International Conference on Machine Learning, pages 330{337, Amherst, MA, June 1993. Morgan Kaufmann

[76] Tatlidede, C. Mericli, K. Kaplan and H. L. Akin, "Q-Learning based Market-Driven Multi-Agent Collaboration in Robot Soccer," Proceedings, TAINN 2004, Turkish Symposium On Artificial Intelligence and Neural Networks, June 10-11, 2004, Izmir, Turkey, pp.219-228
[77] Alejandro Guerra-Hernández, Amal El Fallah-Seghrouchni, and Henry Soldano. Learning in BDI Multi-Agent Systems. Lecture Notes in Computer Science, 3259:218-233, 2004
[78] D. Ancona and V. Mascardi, 2004. Coo-BDI: Extending the BDI Model with Cooperativity. In Selected DALT'03 papers, Springer-Verlag.
[79] Olivia, C., et.al.: Case-Based BDI agents: An Effective Approach to Intelligent Search on the WWW. In: AAAI, Symposium on Intelligent Agents. Stanford University, Stanford CA., USA (1999)
[80] Alejandro Guerra-Hernandez, Amal El Fallah-Seghrouchni, and Henry Soldano, Learning in BDI Multi-agent Systems, Computational Logic in multi-agent systems, pp. 39-44, 2005.
[81] Dhirendra Singh, Sebastian Sardina, Lin Padgham, Stéphane Airiau, Learning Context Conditions for BDI Plan Selection, in proceedings of the 9th COnference on Autonomous Agents and Multiagent Systems: volume 1, pp. 325-332, 2010.
[82] Ferguson, Innes A. (1992) *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. Ph.D. diss., Computer Laboratory, University of Cambridge, Cambridge UK.
[83] Lillis, D., Collier, R.W., Dragone M., O'Hare, G.M.P. An Agent-Based Approach to Component Management, in Proceedings of the 8th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-09), Budapest, Hungary, 2009.