UNIVERSITA' POLITECNICA DELLE MARCHE

Dottorato di Ricerca in Ingegneria dell'Informazione

Curriculum "Ingegneria Informatica, Gestionale e dell'Automazione"

Design of multi-agent robotic infrastructures for the exploration of complex and non-structured environments

PhD dissertation of:

Luca Panebianco

Advisor:

Dott. Ing. David Scaradozzi

Curriculum Supervisor:

Prof. Francesco Piazza

XVI edition - new series

Index

Index of figures

Abbreviations

AGV: Automated Guided Vehicle

AL: Agent Layer

AUV: Autonomous Underwater Vehicle

BDI: Belief-Desire-Intention (Agent)

BT: Behaviour Tree

CL: Connector Layer

COTS: Components-Off-The-Shelf

GUI: Graphical User Interface

MAS: Multi Agent System

MRS: Multi Robot System

OL: Organization Layer

PL: Platform Layer

OsGI: Open Service Gateway initiative

ROV: Remotely Operating Vehicle

SoS: System of System

Chapter 1: Introduction

1.1 Background

The development of autonomous vehicles to perform mission in critical and hazardous environments or situations is a field of study that can be considered fundamental for the whole mankind.

However, in the past, robots had limited processing, sensorial and actuating capabilities, usually requiring well structured environment to allow an autonomous navigation in an efficient and affordable manner. An example of this design is the usage of markers in domotic environments or the exploitation of lines on the floor and basic sensor to detect collisions like the AGVs (Automated Guided Vehicles) in an industrial environment. In the case of advanced intelligent vehicles, like Shakey The Robot [1], that was able to use a complex language to reason and plan the navigation in different indoor scenarios, all the processing was computed, quite slowly, in an external processing unit, that was able to act and sense remotely from the vehicle.

Nowadays, the environments and scenarios where autonomous robots are asked to operate are loosely structured and very complex requiring tailored strategies to navigate. Along with other complex environments, the marine one is considered very important and very challenging at the same time from different points of view, being fundamental to sustain a high number of human activities and life species, while being rich of artefacts from different ages of the human history. In this environment, the communication could be spotty and slow, the visibility reduced and there is no easy way to use odometry to estimate the position in an affordable and precise way. Luckily, along with the increase of complexity of the problem to solve, there has been also an empowerment of processing power, sensors, actuators and communicating devices available at a cheaper price. This process, in the marine

environment, allowed during last decades to allow a transition to the usage of ROVs (Remotely Operating Vehicles) to AUVs (Autonomous Underwater Vehicles).

Another level of complexity is given from scenario that considers cooperation and/or coordination between robots, where different vehicles need to build a society, exchange and share information to perform a joint mission. In literature, to manage this kind of complexity, Multi-Agent System (MAS) theory is cited as a tool capable to manage this issue, by modelling autonomous software components (called agent) that, together, can perform activities than one unique entity wouldn't be able to perform, or with worse performances.

But these agents need to know, reason and plan in a dynamic and unstructured environment, and this kind of actions require a higher level of abstraction, which can be provided by means of a formalization. Because of this, agents could exploit formal models to express concepts such as what they are, what they can do, where they are and how they can cooperate to perform a given mission.

1.2 Motivation and Objectives

This dissertation has two main objectives:

- To analyse different systems and technologies for distributed intelligence through a review of a wide state-of-the-art.
- To design an architecture of a multi-robot infrastructure for the exploration of complex, loosely structured environments by means of the MAS.

The proposed infrastructure, that is the innovative aspect of this dissertation, has two main objectives:

- To express different aspects of Robot and Multi-Robot systems by means of different models.
- To show a layout of a middleware that is able to equip different kind of robots, interpret the proposed models and manage the whole Multi-Robot System.

In order to design this infrastructure, the three years of research had the following phases:

- Analysis and revision of the current state of the art of MAS theory and related technologies to provide a robust background of knowledges
- Analysis and development of different technologies that addressed different aspects of interests such as robot definitions, MASs and system integration.
- Design of models and a middleware to exploit as an infrastructure for multi-robot systems for generic scenarios.

1.3 Thesis overview

This dissertation is organized as follows: in **Chapter 2**, describes the state-of-the-art of different arguments that were investigated. The first topic is about ontologies related to single and multi-robot systems. The second topic is related to agent models, proposing two different classifications and providing some notable examples. The third topic is related to Multi-Agent Systems (MASs) with an introduction to its classification, the usage of this theory in different scenario and some of the most notable MAS frameworks. The last topic is related to the concept of Environment: its characteristics, possible structures, and responsibilities in a Multi-Agent scenario. The **Chapter 3** is tailored to the presentation and analysis of five technologies that have been studied and/or developed inside the LabMACS laboratory during its activity before or during the PhD that contributed, in different measures, to the definition of the models and infrastructure designed and developed in this dissertation. The first one is *DocuScooter*: an innovative device, able to equip different components called Payload, which can provide a model that allows an abstraction of these components. The second one is *OpenFISH*, a modular biomimetic AUV that exploits a MAS to control the Navigation, Guidance and Control (NGC) algorithms and the communication with the underlining hardware. The third technology is the *Home Automation System (HAS)*, a MAS tailored for domestic purposes, where the different apparel can communicate and coordinate to manage different domestic utilities called resources. The fourth technology is a Python and ROS based *MAS for general purpose ASV*. Finally, the fifth study is about the LabMACS Integrated System Architecture, a structuring and modelling paradigm to define technologies and processes used in different missions in the marine environment. Chapters 4 and 5 present two different aspects of the infrastructure developed for a

Multi-Robot System: models that represent different aspects of a robot and multi-robot societies and a middleware able to interpret the models and allow the functioning of different type of robots. In **Chapter 4** three different models are presented. The first model, called *Robot Part Model*, characterizes the hardware that can be connected to the system and that can be used dynamically by the system. The second model, called *Skill Tree Model*, is able to represent the abilities that are enabled for the Robot Parts that are currently equipped in the system. The third model, called *Organization Model* is able to model three different aspects of Multi-Robot Systems: the *Environment* where the robots are situated, the different *Roles* that the robot can play in the system and the *Mission* that they must carry on. In **Chapter 5** the four-layered vertical middleware is introduced. Each Layer (respectively called Connector, Platform, Organization and Agent) is presented and deepened. The main purpose of the infrastructure is to interpret and allow the exploitation of the models introduced in chapter 5. In **Chapter 6** an implementation of the infrastructure and a preliminary simulated application is presented. The simulation reproduces the functioning of the MAS of the ASV presented in Chapter 3. Finally, **Chapter 7** concludes the dissertation by showing some considerations and possible future developments of the infrastructure.

Chapter 2: State-of-the-art

In this chapter, the analysis and study of the state-of-the-art has been carried out by analysing 4 main areas:

- **Robotic ontologies for single and multi-robot systems**: fundamental ontologies about the definitions of robot and multi-robot system will be introduced. The definitions in the rest of this dissertation are related to these ontologies;
- **Intelligent agents**: a rapid introduction of two classifications of agents from two different points of view, from the bibliography, is shown. In the first, the idea is to classify agents by means of general capabilities and ability to express intelligence. In the second one, the classification exploits different concrete architecture layouts that allows to the agent the mapping from the sensors to the actuators. For each architecture, some relevant examples are introduced;
- **Multi Agent Systems**: in this context, a classification for multi-agent system is described and some relevant examples are introduced;
- **Environment**: in this context, a brief review of considerations, dimensions and evolution of the environment is given, in order to gather a good number of requirements about this important component of a multi-agent system.

2.1 Robotic ontologies for single and multi-robot systems

This first part is focused on the ontologies used to define some concepts that will be used during this dissertation. The definitions that will follow are based on the IEEE standard for Ontologies for Robotics and Automation [2]. This standard defines a core ontology that allows the representation of, reasoning about, and communication of knowledge in the Robotics and Automation (R&A) domain. In this reference, the SUMO (Suggested Upper Merger Ontology) ontology is extended to carry out concepts related to this discipline.

SUMO is considered a top-level ontology that defines basic ontological categories across all domain. The SUMO taxonomy can be briefly described in figure 1.

The ontology is presented by means of statements in the SUO-KIF [3](Standard Upper Ontology - Knowledge Interchange Format) language and graphs of some of its main concepts.

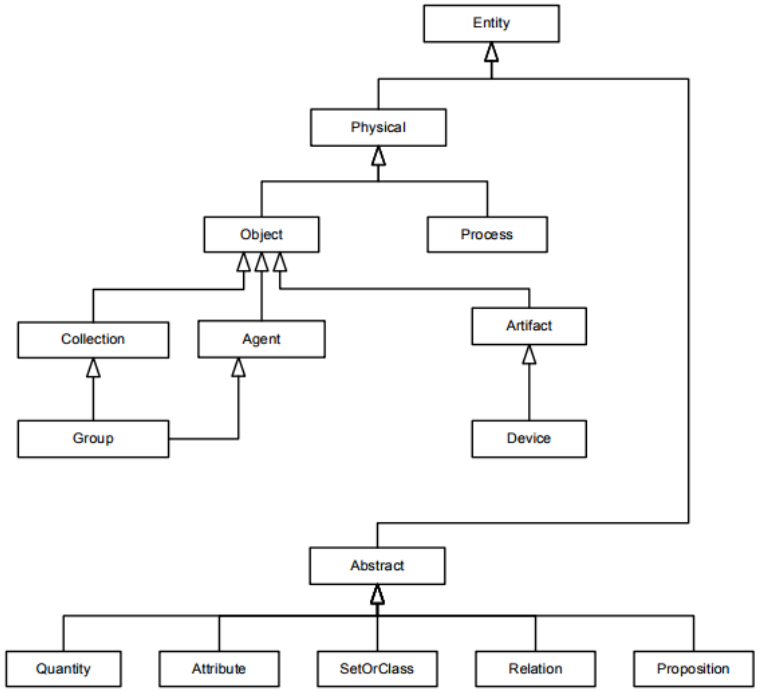

FIGURE 1 SUMO ARCHITECTURE FROM [2]

As shown in the previous figure, the main concept is Entity, that defines an "universal class of individual". Other relevant concepts are:

- **Physical and Abstract**: These two concepts split the main Entity one. They respectively represent entities that they have or lacks a spatial-temporal extension.
- **Object**: an object is a physical entity, and it roughly corresponds to the class of ordinary objects.
- **Process**: a process is a particular kind of physical entity, because it has a spatial-temporal extension but it is related to entities that happen in time and have a temporal part or stages. An example of Process could be a football match or a race. This definition means that SUMO follows an endurantist perspective instead of a perdurantist one. For an endurantist, an object keeps

its identity through time and so, while some processes might change things about it, every part that is essential to it is always present. On the other hand, for a perdurantist, an object is composed of every temporal part it has at all times, so all things about it are indexed in time.

- **Agent**: Something or someone that can act on its own and produce changes in the world.
- **Artefact and Device**: the first is an object that is the product of a making, the second one is an artefact whose purpose is to serve as an instrument in a specific subclass of a process.

Further explanation of other concepts in SUMO are extensively explained in [4].

2.1.1 Robotic ontologies for single robot systems

In the CORA axioms, a *Robot* is a *Device* and an *Agent* at the same time. How this concept is inserted in the SUMO architecture is shown in figure 2.

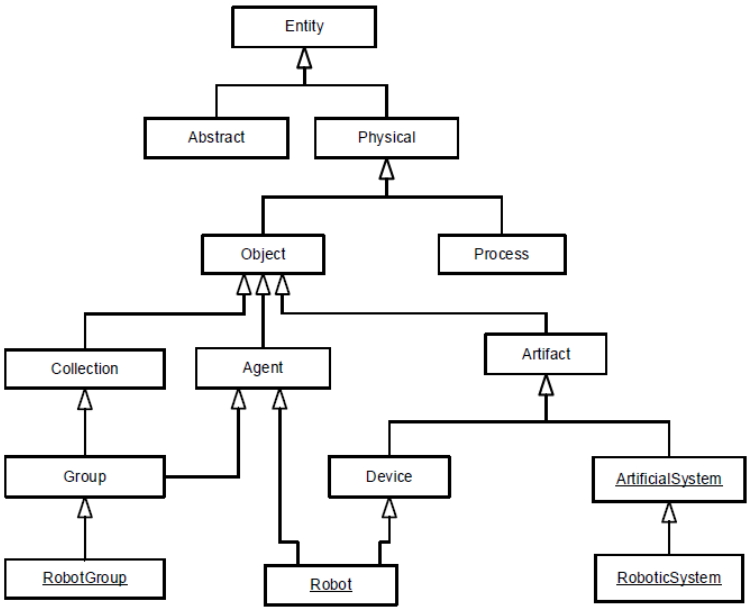

FIGURE 2: CORA ROBOT CONCEPT IN SUMO FROM [2]

Because a Robot is a subclass of Agent, this general definition can lead to misunderstanding, in situation where, for example, a robot requires inputs from an external user in order to execute task. This is a point of ambiguity to differentiate a robot from other machines. This issue is related to the degree of *Autonomy* of the Robot.

CORA assumes that autonomy is related to the particular occurrence of a process while a Robot plays as an agent. CORA allows to a robot to play different roles for different processes.

These levels are:

- **Fully autonomous robot**: with this mode, the robot is able to perform its mission, within a defined scope, without human intervention, while being able to adapt its behaviour to operational and environmental conditions.
- **Semi-autonomous robot**: with this mode, the robot is able to perform task autonomously between human interactions, while the robot acts in an autonomous manner. The human operator can set new task to perform during interactions.
- **Teleoperated robot**: with this mode, the robot needs a human operator to perform its mission. The user receives feedback from the sensorial devices and act through the actuating ones or sets incremental goals during operation, from a location outside the robot;
- **Remote controlled robot**: with this mode, the human operator controls the robot in a continuous way, from a location off the robot and with its direct observation. The robot takes no initiative in this mode and relies in the continuous input from the human operator;
- **Automated robot**: in this case the robot acts as an automaton, with no possibility to altering their actions

The first 4 definitions are directly related to the standard definition of the Autonomy Level of Unmanned Systems (ALFUS) [5] where the different Mode of Operation for Unmanned Systems are described.

Instances of Robot can exhibit characteristics through two relations that are indirectly inherited from Object. The *Attribute* relation allows the robot to get qualitative characteristics by instantiating the abstract class Attribute. The *Measure* relation allows the robot to get quantitative characteristics instantiating the abstract class PhysicalMeasure.

A Robot have other devices as parts, called *RobotPart*. A device can be considered dynamically a RobotPart when attached to the robot. The union of all the RobotParts

are part of the unique RobotInterface device of the Robot. The RPARTS ontology aggregates some of the most general and typical specific types of robot parts. Four typologies of robot part are described:

- **RobotSensingPart**: a part that allows the robot to percept from the environment
- **RobotActuatingPart**: a part that allows the robot act and/or move in the surrounding environment
- **RobotCommunicatingPart**: a part that server as an instrument for robot-robot or robot-human communication process by allowing the robot to send or receive information.
- **RobotProcessingPart**: a part that allows the root to process information

2.1.2 Robotic ontologies for multi-robot systems

In the context of Multi-Robot Systems (MRS) two main sources were taken in account. The first is the same document used to show the taxonomy of Single Robot System [2], that defines briefly this concept, the second reference takes in account two different dimensions to describe MRSs: the first defines different levels of coordination complexity between robots, the second defines different parameters to take in account while developing a MRS.

In the ontology proposed in [2], the concept of *RobotGroup* is introduced, following to SUMO's definition of Group as a "collection of agents". In fact, a RobotGroup is a Group whose member are Robots (figure 3).

The term RobotGroup also includes complex robots. These are embodied agents attached to each other in the same physical structure.

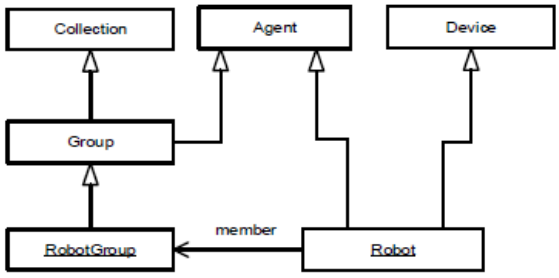

FIGURE 3 : ROBOTGROUP CONCEPT FROM [2]

Robots and other devices can form a *RoboticSystem*. A RoboticSystem is an artificial system formed by robots and devices intended to support the robots to carry on their tasks. Robotic systems might have only one or more than one robot. Robotic systems are partitioned into *SingleRoboticSystems* and *CollectiveRoboticSystems* (figure 4).

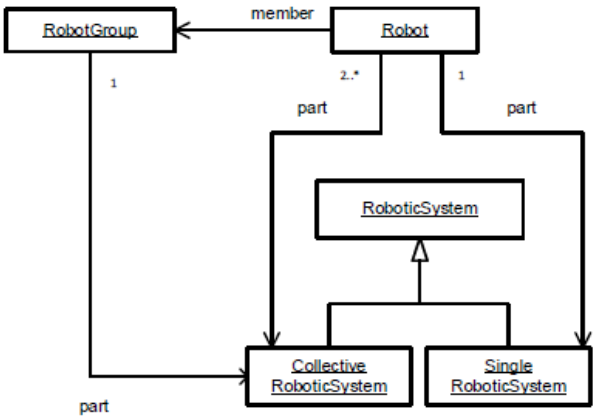

FIGURE 4: ROBOTICSYSTEM CONCEPT FROM [2]

It is assumed that all robots in a CollectiveRoboticSystem collaborate in some way to achieve a common goal. Naturally, each robot may have its own local goal, but these goals must be sub goals of the larger one, i.e., the group's goal. As such, all robots in a CollectiveRoboticSystem must be members of a single robot group. This is the case even in situations where there is no direct interaction between robots, such as in an automated assembling line.

In the second reference [6] a taxonomy tailored to Multi-Robot System is proposed. This taxonomy proposed is characterized by two groups of dimensions: Coordination and System. The first is related to the characterization of the type and level of coordination among robots in a MRS, the latter to define characteristics of essential components that must be taken in account when developing this kind of systems.

The hierarchical structure of the coordination dimension is shown in figure 5.

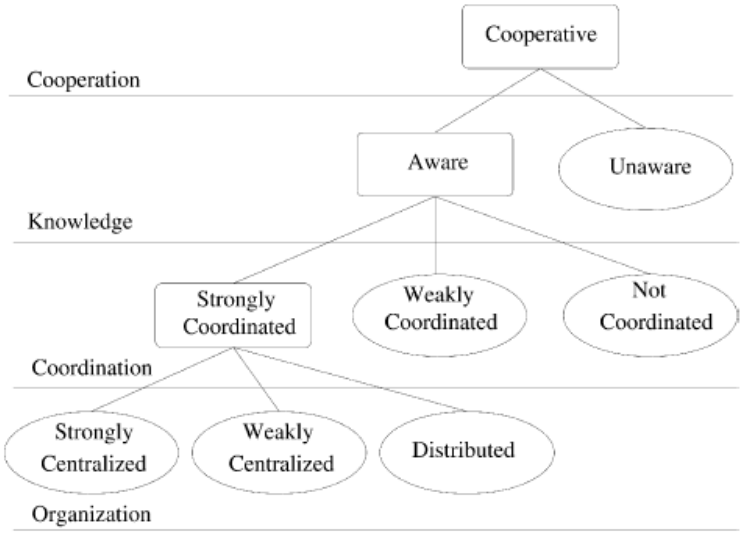

FIGURE 5: COOPERATION DIMENSION FROM [6]

Cooperation Level: this level concerns the ability of the system to cooperate in order to accomplish a specific task. At this level, cooperative and non-cooperative systems are distinguished. This is similar to the previous concept of CollectiveRoboticSystem. A non-cooperative system cannot be intended as a Multi-Robot System so it is not included in the previous schema.

Knowledge Level: the second level of the hierarchical structure is about the knowledge that each robot has about its team mates. Two categories are distinguished: *Aware* and *Unaware* robots. The first kind of robots have some knowledge about other robot of its group, while an unaware robot act without any knowledge of the other robots in the system. Because cooperation is a prerequisite for knowledge, unaware robots doesn't act on their own, but they still must cooperate with other (e.g. by exchanging information) to fulfil the requested tasks.

Coordination Level: the third level is concerned with the mechanism used for cooperation. Following the definition used for Multi-Agent Systems (MAS), coordination is considered as cooperation in which the action performed by each robot considers the actions executed by other robotic agent managing inter-dependencies between the activities of agents in a way that the whole ends up being a coherent and high performance operation [7] [8]. Different levels of coordination are taken in account, based by different coordination protocols that are in use in the MRS. It is possible to have *Not coordinated*, *Weakly coordinated* and *Strongly*

coordinated systems, by means of a missing, weak or strong coordination protocol. Obviously, for robots to coordinate, they need to know which are the team mates, so the presence of an Aware System as Knowledge level is required.

Organization Level: this level is about the way the decision system is realized within the MRS. This level points a distinction in the forms of coordination, separating centralized approaches from distributed ones. Three forms of organization are introduced: Strongly centralized, Weakly centralized and distributed. In a centralized organization, the leader role is present while it is missing in a distributed one. In strongly centralized organization the leader that takes the decisions during the mission duration is predefines. In the weakly one, agents can obtain the leader role in a dynamic way during the mission. In a distributed system, no leader role exists: each robot acts in a completely autonomous way.

The second dimension is the System dimension. This dimension is useful to characterize different features relevant for the development of this kind of system. These features are:

Communication: communication processes among robots are fundamental to obtain cooperation in a MRS by means message exchanges. Two types of communication are taken in account: Direct and Indirect communication. Direct communication is obtained by exploiting dedicated hardware to communicate, while Indirect communication makes use of stigmetry, where the trace left by an action of a robot in an environment stimulates subsequent actions. This technique of indirect communication is defined as a mechanism for universal coordination mechanism [9].

Team composition: team composition can be divided in two main classes: heterogeneous and homogeneous. In the first class, the members of the team are composed by basically the same hardware and control software, while in the second one the robots differ for different for hardware or software capabilities. In relation to the previous definition of robot, robots with different RobotParts are members of a heterogeneous team.

System architecture: this is a fundamental dimension to characterize how the system and its components recover to an unpredicted situation. Two different architectures are introduced: deliberative and reactive. In the first architecture, as a particular event occurs, there are strategies to alter, if necessary, the behavior of all the component of team. In the second one, each robot cope with the event by modifying its behaviour internally to fulfil the assigned task. It is important to notice that this definition does not define which architecture is used for each agent, but how and at which level the overall system copes with particular events.

Team size: this dimension takes in account the number of robots that are involved in the system at the same time.

2.2 Intelligent Agents

The second part of the state of the art is related to intelligent agents. After a brief explanation of what an intelligent agent is, and which are the main differences between a common software, two classic agent classifications are taken into account in this state of the art.

The first one is from Russel & Norvig, introduced in their book "Artificial Intelligence: A Modern Approach" [10]. In this classification, agents are differentiated in five different standardized models. This approach aims to classify agent by their ability to express perceived intelligence, and it is related to different way how software maps perceptions from their sensor to actions through actuators. No strict information is given on how these models need to be implemented in a real machine, that are to be intended as a guide to describe which essential component must be present in an agent in order to express a defined grade of perceived intelligence from an external observer.

The second classification defines five categories of concrete agents, and it's highly inspired to the classification performed by Weiss in his book [8]. This classification aims to differentiate agent by their concrete different implementation of the decision-making process.

2.2.1 Characteristics of Intelligent Agents

In the previous definition of Agent, the schema that is proposed is shown in figure 6.

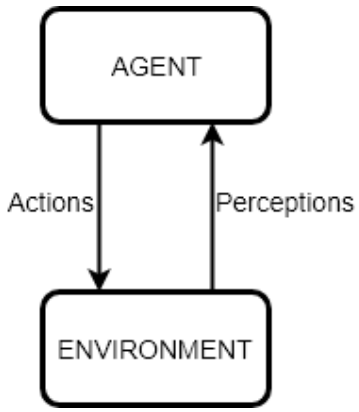

FIGURE 6 : SCHEMA OF AN AGENT

An agent is essentially an entity able to perceive the environment through sensors and act through is actuators in an autonomous way, and it is endowed with the following characteristics:

- **Autonomy**: agents operates without direct intervention of a human operator, and they have a grade of control of their actions;
- **Mobility:** agent could be able to move in an electronic network;
- **Veracity:** agent do not communicate false communication on purpose;
- **Benevolence:** agent do not have conflicting goals and they will do try to do what is asked to reach them;

With this definition, any control system could be defined an agent. A thermostat, for example, is able to control the room temperature by means of a formula that maps the input (the room temperature) to an output (heating on or off).

A necessary augmentation for an agent regards the ability to act in a rational manner in the environment, by exhibiting intelligence. Rationality is the assumption that an agent will act in order to achieve its goals, and will not act in a such way to prevent them to be achieved. An intelligent agent is one that is capable of flexible autonomous action in order to meet its design objectives, where flexibility means three things [8]:

- **Reactivity**: agents can perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives;
- **Social ability**: agents can interact with other agents (and possibly humans) in order to satisfy their design objectives;
- **Proactiveness**: agents can exhibit goal-directed behaviour by taking the initiative in order to satisfy their design objectives.

2.2.2 Agent models

1.1.2.1Simple reflex agents

This class of agents acts on the exclusive basis of the current perception, ignoring the perception history. The processing is conducted by so-called *condition-action rules* of the written as "*IF condition THAN action*". This is considered the most basic type of agent. A representation of this class is shown in figure 7. A consideration is that this kind of agent will be able to success only if the environment is fully observable.

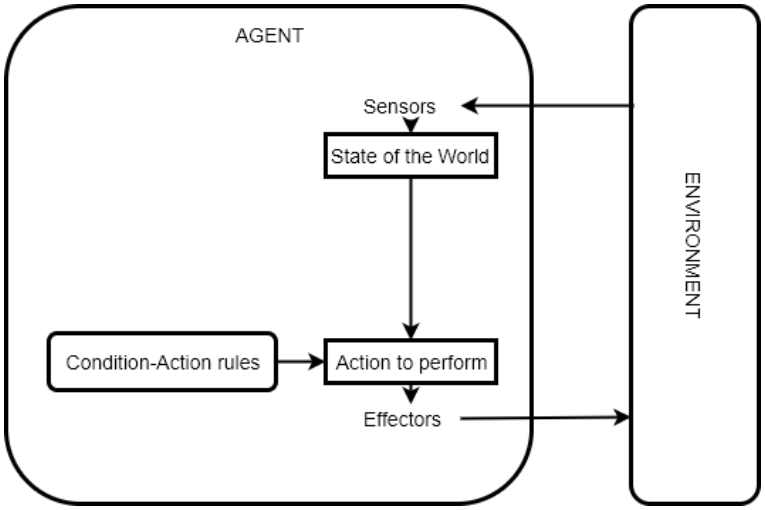

FIGURE 7 : SIMPLE REFLEX AGENT

1.1.2.2Model-based reflex agents

In this class, the concept of internal state is introduced. This kind of agent takes in account different real world-related problem. An example is an agent for autonomous underwater exploration. If the visibility is limited, the agent must keep track of its position and of detected underwater object also if they are currently not

visible. To perform this consideration, the precedent schema of figure 7 is updated in figure 8.

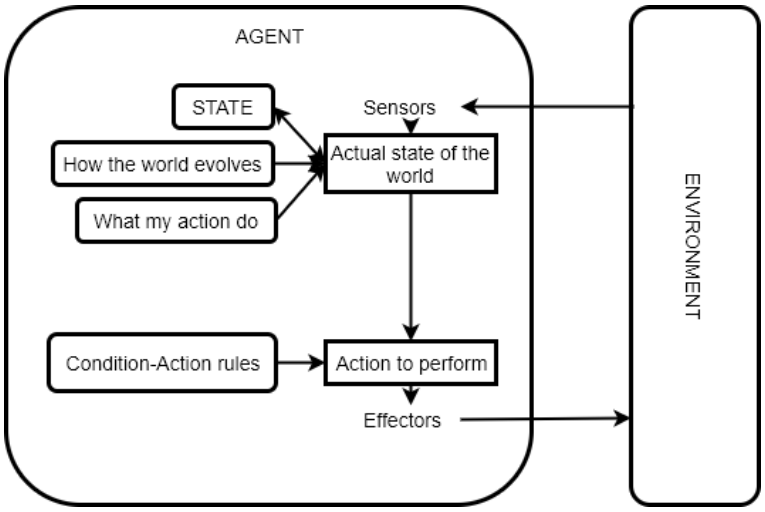

FIGURE 8: MODEL-BASED REFLEX AGENT

The figure shows that the current percept is combined with the old internal state and with other knowledge. In fact, to update the internal state information, the agent must take into account two kind of knowledge: first, information is needed to considerate how the world evolves independently of the agent. Second, the agent should take into account how its action modifies the external world.

1.1.2.3Goal-based agents

This kind of agent programs introduces the concept of Goal, which describes a situation that is desirable. The internal state by its own it is not always enough to decide the actions to perform. For example, for the agent for underwater exploration, the necessity to obtain a situation where the goal "complete the mission" is achieved, will make it plan to obtain this situation, by taking into account which will be the consequences of its action, and which is the best move to do in order to finish the mission. The schema of a goal-based agent is shown in figure 9.

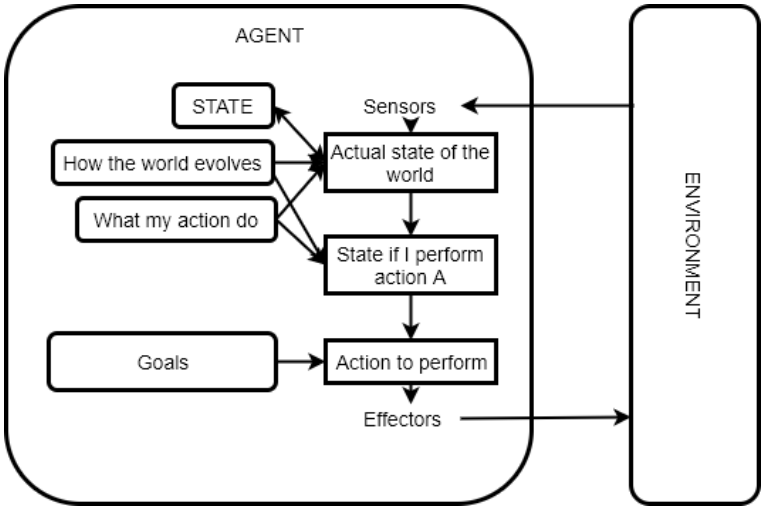

FIGURE 9: GOAL-BASED AGENT

The decision-making process is different from the previous ones and is indeed more complex and less efficient that the two presented before, but is far more flexible. Without the concept of goal, a reflex agent, in order to reach the same objective, should have all the procedures that are able to track the goal hard-coded, and a simple modification of the goal means a modification of part of its rules.

1.1.2.4Utility-based agents

Goals alone are not able to generate high-quality behaviour. For example, there could be many possible action sequences that allow the underwater vehicle to complete its mission, but probably only a few (or just one) are more efficient in some dimension (time, safeness, cost). The goal is quite a "crisp" concept (goal achieved or not achieved), while usually more soft and measurable criteria is needed to evaluate the performance of different world states. This criteria is called Utility. The Utility is a function that maps a state into a real number, which defines the associated degree of desirability of the actual situation. The schema of a utility-based agent is shown in figure 10.

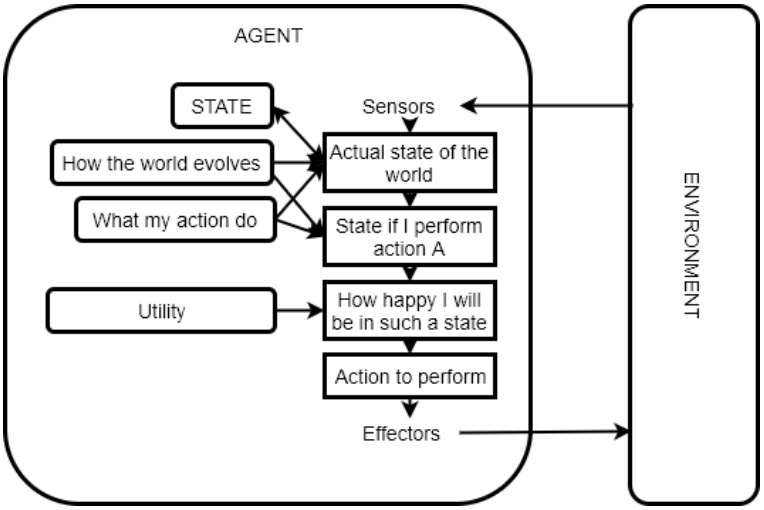

FIGURE 10: UTILITY-BASED AGENT

The use of a utility function instead of goals have some advantages specifically in two kinds of case, where it allows to perform rational decisions in comparison to a goal-based architecture. The first case regards conflicting goals: when only some of them could be achieved at the same time (for underwater case an example could be vehicle energy efficiency and speed), the utility function could be able to perform a trade-off between these two conflicting goals. The second scenario is related to goals that the agent cannot be sure to have achieved, but there is a certain degree of uncertainty that it is reached totally. Utility in this case allows to weight the likelihood of success against the importance of reaching that goal.

Because the world, in real world scenario, is usually partially observable and stochasticity are ubiquitous, decision making is made under uncertainty. Therefore, a rational utility-based agent chooses the action that maximize the *expected utility* of its actions.

1.1.2.5Learning agents

Learning allows an agent to improve it performances and to operate in an initially new unknown environment and to become more and more capable than its initial knowledge alone might allow. As shown in figure 11, the structure of a learning agent has a very different structure and is possible to describe it as an ensemble of four conceptual components.

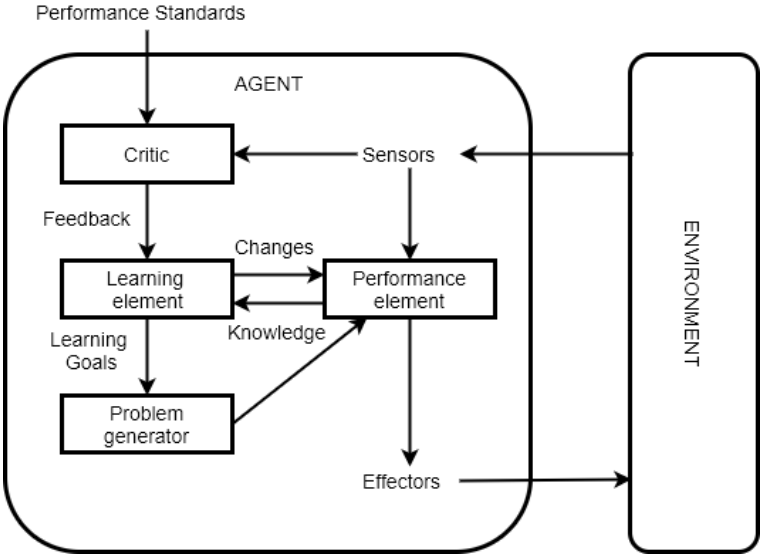

FIGURE 11 : LEARNING AGENT

The Performance element is what was until now considered to be an entire agent: an entity able to percept the environment through its sensors and to select an external action to perform through its effectors. The Learning element is the component responsible to perform improvement. It uses the feedback from the Critic in order to determine how the agent is behaving and how the Performance element should be modified to perform better over time. This component is necessary because perceptions on their own give no clue to the agent about how well it is behaving or if it is succeeding in its task with respect to a fixed performance standard. In the previous schema, it is possible to notice that the Performance Standards are considered fixed and external to the agent. The last component is the Problem generator. It is responsible for suggesting actions that will lead to new and informative experiences. Without this component, the learning element will provide the best actions for the knowledge that it has instead of finding new ways to act that are more efficient than the already known ones.

2.2.3 Agent Architectures

Each architecture represents a different way to implement how the agent is able to map its perception to its actions. It is possible to define at least five different architectures:

- **Logic based/Deliberative reasoning architectures**: where decision making is performed by means of logical deduction;
- **Reactive architectures**: where decision making is performed by some form of direct mapping from situation to action;
- **Belief-Desire-Intention architectures**: where decision making is performed by the manipulation of data structures representing beliefs, desires and intentions;
- **Layered/Hybrid architectures**: where decision making is performed by different software layers, each of them with a different level of abstraction of the environment;
- **Cognitive architectures**: where decision making is performed replicating human cognitive phenomena;

For each architecture, a rapid explanation of its strength and weakness is given, then the most notable and relevant examples of each architecture are presented.

2.2.3.1 Logic based/ Deliberative reasoning architectures

In the traditional and first approach to building artificial intelligent system (known as symbolic AI), suggests that an intelligent behaviour can be achieved by giving to the system a symbolic representation of the environment and its desired behaviour, and then syntactically manipulating this representation. In the case of Logic based/Deliberative reasoning architecture, these symbolic representations are expressed by means of logical formulae, and the syntactic manipulation corresponds to logical deduction, or theorem proving.

This approach has pros and cons. As positive aspects, the whole process used to manipulate the logic formulae follows rigid and proven theorems, and, if a decision is taken from the agent, how and why was chosen is logically explainable and provable. Furthermore, the logical semantic used in these architecture is simple and elegant. By the other hand, this approach has a lot of problems. As instance, when the first examples of these architecture were investigated it was obvious that these architectures were anything but instantaneous and, in some cases (if the agent uses classical first-order predicate logic to represent the environment, and its rules are sound and complete) there is no guarantee that the decision-making process will

terminate. Moreover, a symbolic representation of the actual real-world environment is very complex and too cumbersome for resource limited systems (space and computation power). Finally, since this system aims to provide the best solution to solve the problem each time, any change in symbolic representation of an element of the environment (for example if an obstacle is detected) requires that all the process solving (theorem proving) mechanism must be performed again. This is an issue especially when the state of the world changes faster than the agent can make decisions.

Three examples of this architecture will be presented: Concurrent MetateM, based on temporal formulae, ConGolog, based on situation calculus and then Situated Automata, based on a total different approach, in which the modelled behaviour is directly compiled in the agent.

Concurrent MetateM

Concurrent MetateM [11] is a multi-agent programming language that uses temporal formulae as logical language. These agents are able to communicate through broadcast message passing. Each concurrent MetateM agent has two main components: an interface, which defines how it is able to interact with the environment or other agents and a computational engine, which defines how the agent will act through the definition of a set of temporal formulae. Three sets are described:

- **PML$^+$**: formulae referring to the present or future called commitments;
- **PML-**: formulae referring to the past called history formulae. It is a collection of message received and action performed by the agent during its whole life-cycle.
- **PML±**: formulae in the from past => future called rules;

The history formulae and the commitment together form the agent's internal state,

Each agent is programmed by giving it a specific temporal logic called First-Order MetateM Logic (FML) and its execution is a cycle that: updates the history of the agent by receiving messages, check which rules can fire by comparing past-time antecedents of each rule against current history and then jointly execute the fired rules together with any commitments carried over from previous cycles.

ConGolog

In [12] the authors propose a particular methodology to avoid computationally demanding plan synthesis in real-world scenario, where dynamic and partially observable environments conduct to very complex systems. The alternative approach is based on high-level program execution [13]. Instead to search for a sequence of action that are able to take the agent from the initial state to the goal state, the idea is to find a sequence of domain-dependent actions that are part of a legal execution of a high-level non-deterministic program. The objective is to conveniently express with these high-level programs the most part of the agent needs and, doing this, by easing computationally the execution of the program in comparison to the corresponding planning task.

The language used to define these high-level non-deterministic program is called Concurrent Golog (ConGolog) that is based on situation calculus [14]. This language is an evolution from the previous Golog language [13]. ConGolog is able to model goal-oriented agent controllers while concurrently monitoring and reacting to condition in their environment.

Situation Calculus is a first-order language (with some second-order features) for representing dynamic domains. A world history, that is a sequence of action, is represented by a first order term called a situation.

Situated Automata

Another approach was introduced by Rosenbach and Kaelbling with their Situated Automata architecture. Instead of syntactically define and manipulate the world through logical languages on order to provide a formal planning method during the runtime of the agent, in this architecture the logical specifications are compiled to provide a light, more efficient decision-making process. Situated automata has its fundamentals in a formal semantic of embedded computation [15]. It gives a specification of the information content of the internal states of a machine in terms of the external states of the environment in which that machine is embedded.

An agent, in this architecture, is seen as an entity capable of performing a transduction between the inputs that are acquired from the environment to a

stream of actions that interact with the environment. This transduction is modelled as a finite state machine by means of fixed sequential circuit. The formalism of the language allows to compile this software in a digital machine, which must satisfy the declarative specification. The schema of the circuit that implements this theory is shown in figure 12.

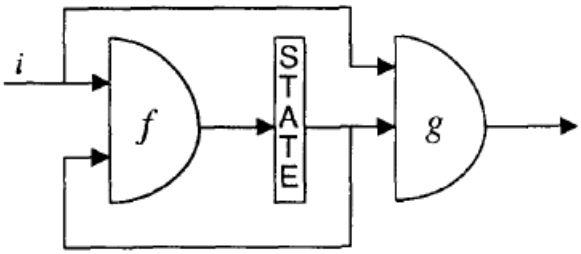

FIGURE 12: SITUATED AUTOMATA CIRCUIT SCHEMA FROM [15]

A sequential circuit (and so the agent) can be decomposed into two functions: function *f* calculates the new state of the internal circuit and can be considered as the perception module of the machine. Function *g* calculates its output as a function of the old value of the state and an input function *i*. this latter function can be considered a free-state action component, capable to decide at each instant which action to take and can be considered the action module of the machine.

Two different languages manage the development of these two models: RULER [16](perception) and GAPPS [17] (action). RULER synthetizes machines able to track semantically complex conditions in the environment by means of constant-time update circuitry. GAPPS can maps a top-level goal and a set of goal-reduction into an efficient parallel compiled program able to make a transduction from a stream of input incoming from the perception component of the robot to a stream of output actions.

2.2.3.2 Reactive architectures

A reactive architecture approaches the problem of defining the architecture of an agent by taking in consideration the necessity of a more robust and real-time behaviour. In fact, researches in the mid-to-late 1980s started to investigate alternatives to the previous symbolic AI paradigm. Some emergent ideas were:

- Rejection of symbolic representations, and decision making based on the syntactic manipulation of such representation
- The idea that intelligent, rational behaviour is linked to the environment an agent occupies, a product of the interaction the agent has with the environment
- The idea that intelligent behaviour emerges from the interaction of different simpler behaviour

Some practical limitations of the deliberative reasoning architectures were appointed by Brooks [18] while introducing his subsumption reactive architecture like:

- **Management of multiple goals**: usually an agent will have multiple (sometimes in contrast) goals to achieve like reach a specific point while avoiding obstacles.
- **Managing of multiple sensors**
- **Robustness**: the agent must adapt and cope to malfunctioning to its sensors or actuators or to drastic changes of the environment.
- **Extensibility**: the agent must cope with sub sequential modification to its structure, also allowing the minimum effort to change a single component of the system.
- **Real-Time behaviour**: the problem of high computational effort of problem solving of logic architectures is not good for real-world/real-time scenarios.

In opposition to the traditional decomposition of a mobile control system at that time, commonly deigned as figure 14 shows, this type of architecture is decomposed as a series of horizontal-stacked layers as figure 13 shows. It is possible to notice that, in this latter kind of architecture, each layer can access directly to the sensors and act through the actuators.

Another important aspect is the lack of an explicit model of the environment: all the useful information to perform the decision-making process must be acquired from the current local environment.

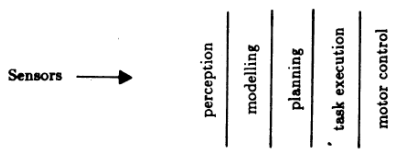

FIGURE 13: HORIZONTAL ARCHITECTURE FROM [18]

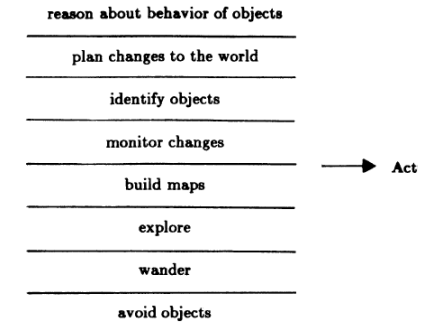

FIGURE 14: VERTICAL ARCHITECTURE FROM [18]

In summary, this architecture since its beginning has proven to be simple, robust, easy to implement and to embed, but there are some problems still partially or totally unresolved as: lacking an environment model or history, that forces the agent to take decision based on an actual local knowledge and of some advanced features like learning. Other problems are related to the actual development of these agent as challenges related to the development of complex agents composed by a high number of layers: these agents could have an emergent behaviour difficult to build without a proper methodology and to track while many layers are interacting between each other's.

Two examples of Reactive Architectures are proposed. The first is the most famous one: Brooks' Subsumption Architecture, the second is the Agent Network architecture, that is based on the relationship of different modules expressed Subsumption agents. Other notable examples are the PENGI system [19] and Firby's [20] Reactive Action Packages.

Subsumption Architecture

The subsumption architecture has been proposed by Brooks in [18] and is known to be the best pure reactive architecture. This architecture has been proposed due to the dissatisfaction with the performances of the representation-inspired robots in dealing with the real world like Shakey the robot [1].

The subsumption architecture is a hierarchy of task-accomplishing behaviour each one in a so called different "layer of control", each one with a different competence. Its underling schema is shown in figure 15.

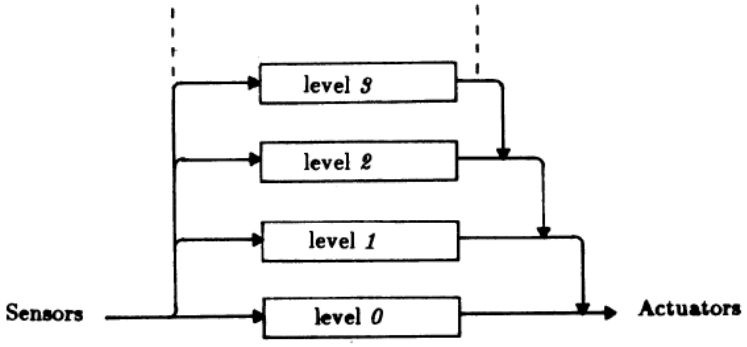

FIGURE 15: SUBSUMPTION ARCHITECTURE FROM [18]

As can be seen, this architecture allows to all layers to access to the sensor's and actuator data and operate in parallel. The low-level layers take in account more primitive typologies of behaviour (like avoid obstacles and wander) and are designed to react fast to environment changes while high-level layers have the duty to control the system towards overall goals. The low-level layers have precedence over further layers.

Each layer has its own simple and unique goal and it can fulfil it without the aid of other layers, enabling a modular view to robot programming. This allows easier testing and debug of each functionality of the system, instead of testing a bigger system that take care of all aspects of the robot. There are two aspects of the modules of each layer to be considered: the internal structure of the module and the way they communicate.

Each module is developed as an Augmented Finite State Machine (AFSM) with a series of input and output lines. The augmentation is composed by added instance variables to hold data structures. A schema of a ASFM is shown in Figure 16. The ASFMs are synchronized and receive perceptions from the sensors at the same time.

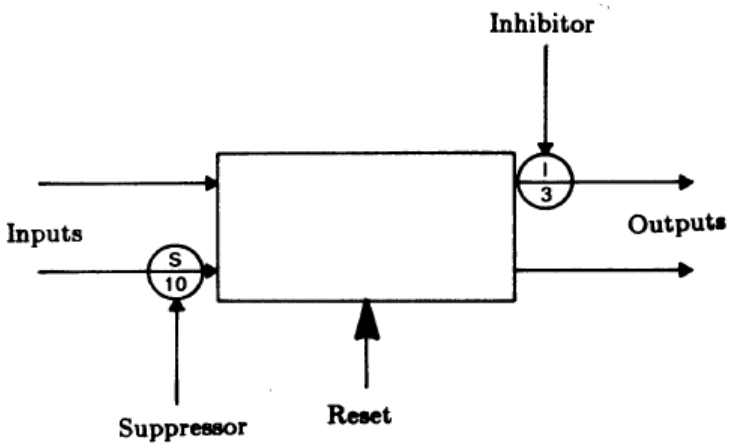

FIGURE 16: AUGMENTED FINITE STATE MACHINE FROM [18]

The communication between modules is performed by connecting input and output lines of different modules through wires. A wire has a source and a destination. In addition, outputs can be inhibited and inputs suppressed for a certain amount of time. These mechanisms allow to override and alter the external behaviour of a module.

If an input is suppressed. Any signal received will be substituted with the replacement provided by the suppressor. In a similar way, if an output is inhibited, any messages sent by the module will be lost. Both inhibitor and suppressor terminals have a time constant.

ANA

The Agent Network Architecture (ANA) that is based on the studies of Maes [21] and it proposes an architecture to develop autonomous agents. It is inspired on the Minsky's Behaviour-Based AI [22] and by Brook's subsumption architecture [18].

The architecture is divided in different competence modules, which correspond to different Subsumption behaviour. Two types of modules are defined: the first is the Action Module, which allows to the agent to perform some physical action in the environment, and the Belief Module, which allows to adopt a belief for a limited time. Each module is described by different parameters: a list of conditions to be met, of added and removed conditions and an activation level. The last parameter indicates how much a module is relevant in the current context and is used to define which module will take control of the agent.

The modules are divided in groups that are incompatible among each other. An action module is incompatible with another one if it has a shared resource, so they can't act at the same time. In a similar way, Belief Modules are incompatibles if they express contradictory belief. These modules are considered as black box, and more implementation of them are proposed (some proposed examples are theorems, neural nets, a piece of software).

An important task that the architecture must perform is the selection of the actual running module. This problem involves the use of a spreading activation network to assign an order to modules to run. The description of each module allows to build a net of predecessors and successors. If a module accumulates enough activation energy through its connected links it acts (if it is an action module) or its beliefs are adopted (if it is a belief module). More specifically, there are two components to the spreading activation dynamics. First there is the external input/output of activation energy: goals, data acquired from sensors and protected goals (goals achieved but that must remain achieved during long term). Secondly the internal spreading of energy among competence modules: a non-executable module can increase the energy associated to its predecessor, an executable module increases the level of its successor and every conflicting module lowers the energy of its conflictors. An example of this network is shown in figure 17.

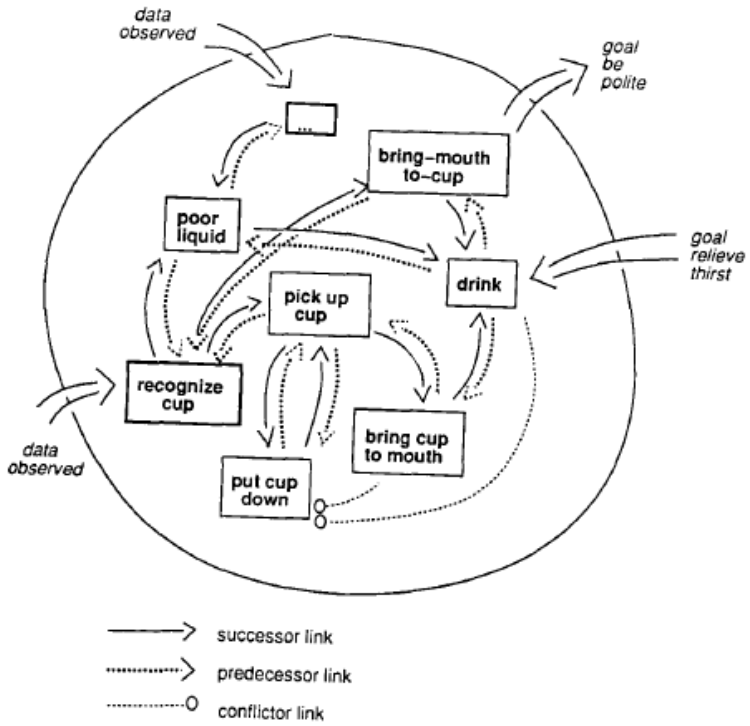

FIGURE 17 : EXAMPLE OF SPREADING ACTIVATION NETWORK FROM [21]

For each loop, the external and internal activation is computed, a decay function is applied and finally the strongest module that has an energy higher than its activation level is selected.

2.2.3.3 Belief-Desire-Intention architectures

The Belief-Desire-Intention model is based on Dennett's theory of intentional systems [23] and on Bratman's theory of human practical reasoning [24] that is a reference to explain future-directed intentions. Practical reasoning is about deciding, moment by moment, which action to perform in order to reach our goals. This activity is composed by two different stages: the first regards the decision about which state an agent want to achieve (called deliberation) the second regards how to reach these states (called means-end reasoning). This process is different from the theoretical reasoning that derives knowledge or reaches conclusions by using one's beliefs and knowledge.

In Bratman's theory Beliefs are facts regarding the environment. Among those there can also be inference rules that can lead to the acquisition of new beliefs. Desire is the so called motivational state: the goal that the agent want to achieve. Intention is

the deliberative state of the agent, the commitment of the agent to plan and try to reach the chosen desires.

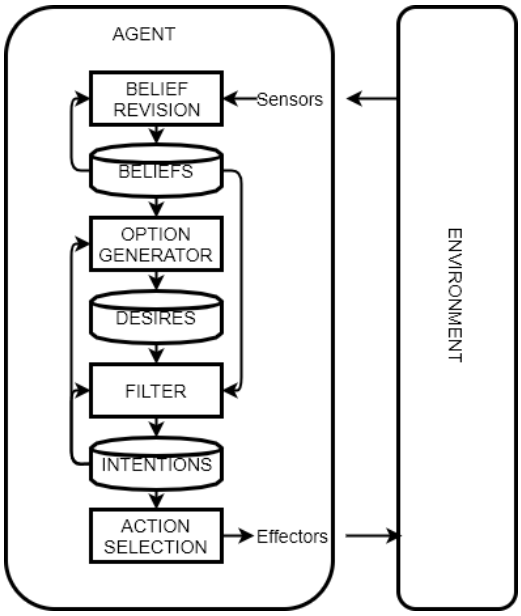

FIGURE 18 : A BDI AGENT ARCHITECTURE

There are some basic functions and structures in these architecture, shown in 18: A Belief Revision Function that determines new beliefs starting from the perception of the environment, a set of current Beliefs, an Option Generator Function that determines the options available to the agent based on the current beliefs and intentions(it represent the agent's means-end reasoning process), a set of Desires, a Filter Function that determines the agent's intentions based on current beliefs, desires and intentions (it represents the agent's deliberation process) a set of Intentions and an action Selection Function that determines an action to perform based on current intentions.

The resulting BDI architecture is very intuitive indeed, by using processes of "decide what to do" and then "how to do it" that are common to human behaviour as the concept of belief, desire and intention. Then there is a clear functional decomposition, which indicates what sorts of subsystem are required to build. The problem that remains is to find a way to implement these functions efficiently.

An example of BDI system is proposed: Georgeff and Lansky's PRS.

PRS

The architecture of a PRS [25] (Procedural Reasoning System) agent is shown in figure 19. Some of its features are: reactivity to the environment (the agent is able to easily select a new plan if new relevant events happen), reflection (the agent is able to reason about its internal state), partial planning strategy and use of procedural knowledge.

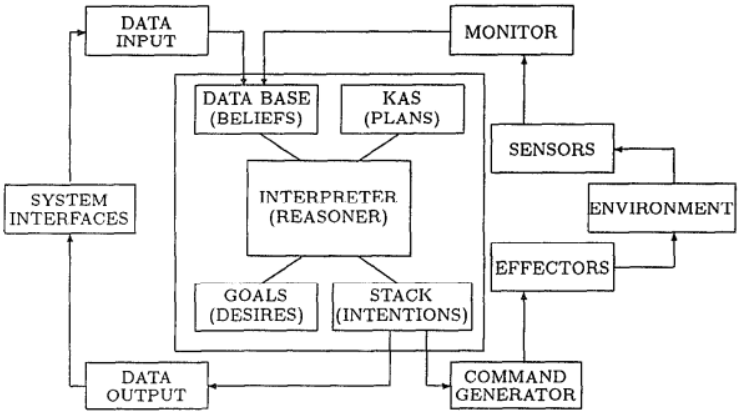

FIGURE 19: PRS ARCHITECTURE FROM [25]

The architecture consists of a central core composed by five components: a database containing current belief describing what is believed as true at the current instant of time. They are defined as facts representing static properties of the application domain. To express them a language based on first-order predicate calculus is used. Then the architecture is composed by set of current goals (the desires of the BDI model), representing desired behaviour rather than static world states that the agent tries to achieve. Then there is a set of plans called knowledge areas (KAs) describing how certain sequences of actions and tests may be performed to achieve given goals or react to different situations. Each KAs consists of a body, which describes the steps of the procedure, and an invocation condition, that specifies under which situations the KA is useful. The stack of the intentions of the agent and contains all the currently active KAs. Finally, the interpreter (or reasoner) is the main component that can manage and manipulate these previous components.

Its functioning is an interleaving of plan selection, formation and execution. Essentially, at a particular time, certain goals are active and beliefs are present in

the data base and, in accord with these condition, only some KAs will be applicable. The interpreter will choose which KA to execute and will apply the consequences of the choice by means of new sub-goals or beliefs derived. If a new goal is pushed in the goal stack, it will select a relevant applicable KA and will start to execute it. If a new belief is added in the data base it will perform consistency maintenance with the current KA or will select a new, more relevant KA if necessary.

This architecture was initially used to equip a robot named Flakey [26] to perform tasks such as autonomous navigation through hallways, corners and avoid obstacles and then it was used to implement malfunction-handling tasks [27].

2.2.3.4 Layered/hybrid architectures

Hybrid architecture interleaves the pros of the two previous architectures integrating the deliberative and reactive architecture in a unique one. This main reason of this approach is to exploit the reasoning, planning, goal handling and abstraction capabilities of deliberative architectures and the real-time, robust responsiveness of reactive architectures. Hybrid architecture can be subdivided in vertical and horizontal layered.

In horizontal layering (figure 20) the software layers are each directly connected to the sensory input and the actuating output. Each layer acts as an agent on its own, usually at different levels of abstraction, and are able to produce suggestions to action to perform. It has the advantage to be conceptually simple: if an agent must exhibit different types of behaviour, it is possible to implement more horizontal layers. By the other way, these layers concurrently compete to make its own decisions, so it is necessary to implement some sort of “control” mechanism. This latter is usually called mediator, which acts as a bottleneck, limiting the possibility to have actual parallelism thanks to autonomous layers.

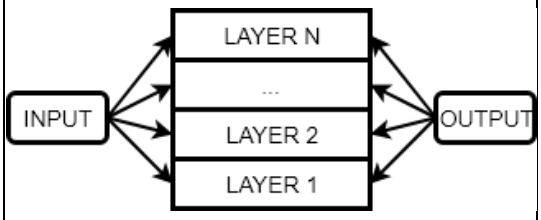

FIGURE 20 : HYBRID HORIZONTAL ARCHITECTURE

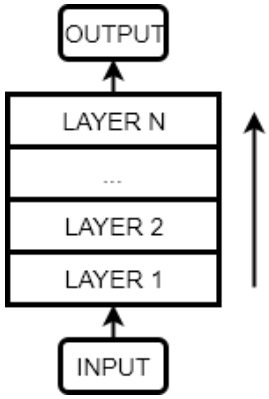

FIGURE 21 : HYBRID VERTICAL ONE-PASS ARCHITECTURE

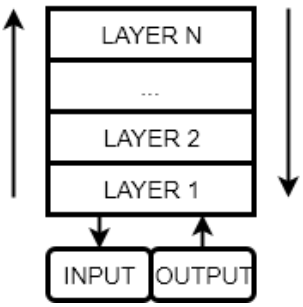

FIGURE 22 : HYBRID VERTICAL TWO-PASS ARCHITECTURE

In vertical layered architectures, by the other hand, because they are composed as a stack of layers, just one of these layers can have direct access to inputs and outputs. More precisely, vertical structure can be subdivided in two groups: one pass architectures (figure 21) and two pass architectures (figure 22). In the first, control flows sequentially through each layer (so, for example, two separate layers manage the sensory input and the actuating output) while in the latter, the information passes from low-level layers to high level and then flows back. This partially solves the problem of complex mediator for horizontal layered architecture.

Different kind of architectures are proposed: four two-tiered vertical architectures (SSS, 3T, AURA and InteRRaP) in a sort of complexity order, and one horizontal layered architecture (TuringMachines).

SSS

SSS [28] is an architecture introduced by Jonathan H. Connell. Its schema is shown in figure 23. It is a 3-layered vertical two-pass hybrid architecture. The layers are called Servo, Subsumption and Symbolic (the three layers compose its acronym). It was tailored to develop a system able to fully control a robot for indoor navigation.

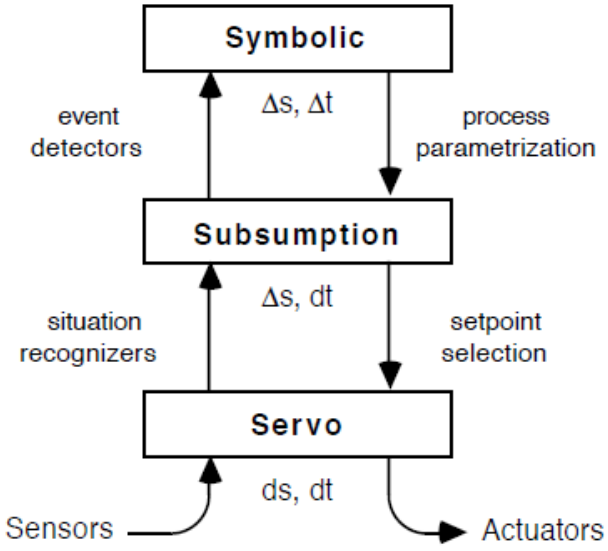

FIGURE 23: SSS ARCHITECTURE FROM [28]

The architecture exploits the features of three different technologies: the servo layer allows to process real-time signals by interacting directly with sensors and actuators; the subsumption layers, by means of Brooks Subsumption architecture allows to act quickly to events occurring such as obstacle detection by acquiring the sensor signal in a discrete manner; the last layer allows to have a model of the world where the robot is situated and take deliberations on order to fulfil the given goal.

Another aspect to be taken in account in this architecture is the connections between each layer that exploit different technologies. In the architecture when the interface is between a lower level to an upper one is called "sensor" interface, by the other way it is called "command" interface. The command connection between the subsumption and servo layers is performed through the setting of set points for the lower layer while the sensor communication is performed by means of matching filters, which can filter significant values of the data acquired from the Servo level. The sensor interface between the Symbolic and Subsumption is accomplished by a mechanism that generates events that signals to the upper layer if specific conditions are met. Finally, the command interface between the uppermost layer and the middle one, allows to enable and disable each behaviour selectively and parametrize them.

It is possible to notice that, although this is a hybrid architecture, there is a strict information passage between the three layers, different layers can access to

incrementing abstract versions of the same data. The three layers in the system come from progressively quantizing first space then time.

The resulting robot, called “TJ”, could automatically map indoor buildings and navigating through them. The tactical navigation (moment-to-moment control of the robot) was carried out by the Servo and Subsumption level using odometer data, while the strategic navigation (planning about what to do next) was carried out by the Symbolic level. This level could keep in memory a coarse map of the surroundings by means or IR proximity detectors. The average speed during the real-world test was 32 inches per second.

3T

3T is a 3-tiered vertical two-pass hybrid architecture (this characteristic gave its name) developed during an 8-year work [29]. An important aspect is that, along with the architecture, specific software to design and develop features for specific robots or mission has been developed. An example of a robot exploiting this architecture is shown in figure 24, while the generic intelligent control architecture is shown in figure 25. It is possible to easily notice the three different layers: a dynamically reprogrammable set of skills with a skill manager to coordinate them, a sequencer that can activate and deactivate skill to accomplish specific tasks and a deliberative planner that takes on accounts constraints such as goals, timing and resources.

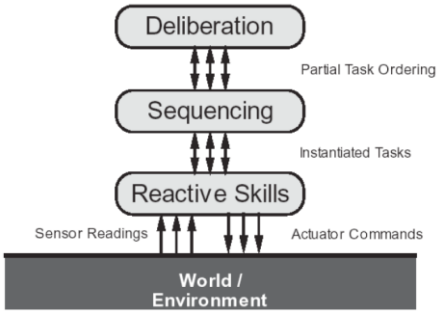

FIGURE 24: 3T ARCHITECTURE FROM [30]

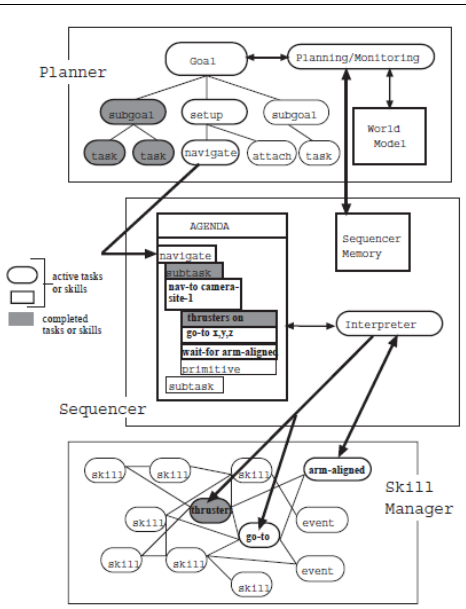

FIGURE 25: EXAMPLE OF A ROBOT USING 3T FROM [29]

Regarding the architecture, the skill manager can interface the set of situated skill to the rest of the architecture. A situated skill is considered the connection between the architecture and the external world. The term situated skills is intended to denote a capability that, if placed in the proper context, will achieve or maintain a specific state in the world. The modular approach to skill development, which must be followed for each different robot that implements 3T, forces a standard interface among the skills and the sequencer. This representation includes information about: input/out specifications, a computational transform, an initialization routine and an enable/disable function. Each skill is developed to be totally independent from others. These are particular skills that take input from other skill and notify the sequencer if a desired state is detected.

The sequencer is the second tier of the architecture and it is implemented with a RAP (Reactive Action Package) interpreter [31]. A RAP is a planning system that uses decomposition rules and an interpreted language to represent sequences of action in order to accomplish the mission' task. It is notable to notice that usually RAPs allow different approach to solve a specific task by means of the actual knowledge of the environment or of the capabilities of the vehicle. The interpreter runs in a continuous loop installing new goal in its agenda, removing old ones that

have been achieved or failed, enabling/disabling skills, and watching for response events from event skills.

The planner is specific for tasks that are difficult to specify as a list of common robotic skills. The planner used in 3T is AP [32]. The AP planner can also to reason about uncontrolled agents (like humans or even nature) by using a counter-planning mode to reason about how preconditions or during conditions in a plan might be negated by an uncontrolled agent, thus thwarting the plan.

3T deals with failure at three levels: environmental variation in the skills, variation in routine activity in the RAPs, and variation in time and resources in the planner.

This architecture has been used in different projects: a robotic wheel chair [30] using the skill manager and an abstraction of the sequencing layer, three different robots with different following and approaching tasks using the skill manager and the sequencing layer, the full architecture has been used for a simulation of a three-armed EVA Helper/Retriever robot to carry out tasks around a space station and finally the full architecture has been implemented for a robot project, with the task of simply running errand.

AuRA

AuRA [33] (Autonomous Robot Architecture) is a two-tiered vertical two-pass hybrid architecture, its schema is shown in 26. Its development began in the mod 1980 as a hybrid approach to robot navigation, dividing the deliberative and the reactive aspect of a robot. It is considered the first robot navigation system with this kind of approach.

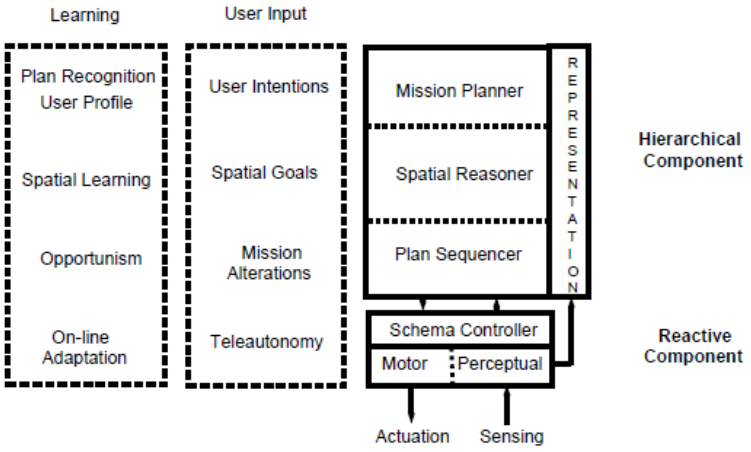

FIGURE 26: AURA ARCHITECTURE FROM [33]

The deliberative aspect is represented by the hierarchical component, while the reactive part by the reactive component. The hierarchical component is divided in three modules: the mission planner, the spatial reasoner and the plan sequencer. The first component has the duty to interface with a human operator to receive the mission/prompt its status. The second can update and maintain a local map of the area and plans paths to execute the requested mission. Finally, the last component translates the single paths received from the spatial reasoner in a set of motor command. Then these commands are sent to the robot to execution. This communication divides the deliberative and reactive component.

The functioning of the hierarchical component is bottom-up: if the plan sequencer fails to plan a set of action to get to a certain point of the path, the spatial reasoner is invoked to plan another path to the goal. If it is impossible to find a feasible path to reach the goal, the mission planner is then invoked, informing the operator of the difficulty found and asking for reformulation or abort of the mission.

In the reactive controller, the schema manager is responsible to monitor and control the different motor and perceptual schemas during run-time. Each motor schema is associated with a perceptual schema capable of providing the stimulus for it. Once reactive execution begins, the deliberative layer is invoked only if a failure is detected. A failure could be denoted by the lack of progress or by the reach of a timeout.

InteRRaP

The InteRRaP architecture [34], is an evolution of the RATMAN architecture [35]. The overall agent schema is shown in figure 27.

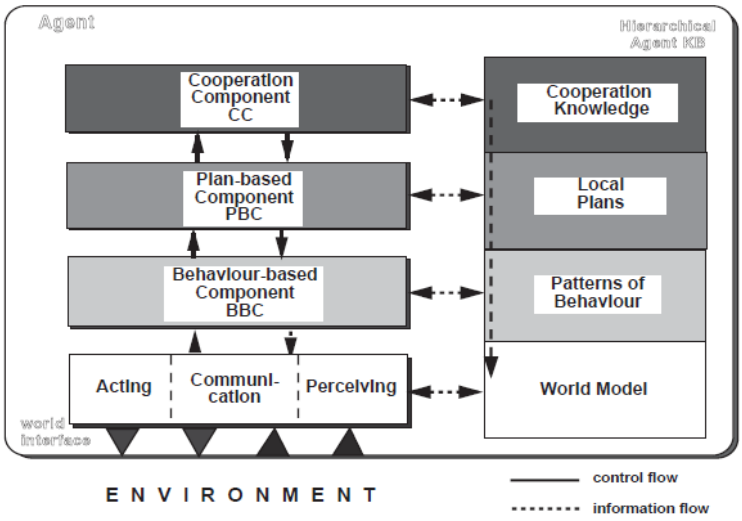

FIGURE 27: INTERRAP ARCHITECTURE FROM [34]

It is clearly divides the knowledge and functionality aspect of the agent with its two-pass hybrid vertical architecture composed by two components:

- Multi-stage control unit (left side of the image).
- Hierarchical agent knowledge base (right side of the image)

In the bottom of the multi-stage control unit there is the World Interface (WIF) that is the facility for agent perception, action and communication. To be effective, the information that the agent receives or perceives must be filtered and adapted into an explicit representation, which is stored in the World Model (WM).

The second layer is the Behaviour-Based Component(BBC). It allows to react fast in certain critical and routine situations. This reaction is done by selecting and managing the execution of one or more Pattern of Behaviour (PoB). This component has access to the world model (because its decision and action are closely linked to the real world) and to a storage of available PoBs.

By one hand, PoB represent a basic reactive problem-solving facility of an agent (avoid obstacle), by the other hand, they allow to describe pieces of procedural knowledge, mechanisms that are not represented in a declarative manner, but are

basically procedures as routines procedures, action that can be taken without a deep reflection or planning but that cannot be considered as a reactive behaviour (power on engine).

The data associated to a PoB are name, priority, description, situational context of the pattern (condition that must be verified to be able to execute it), the mental context of the pattern (defines the goals that are affected by the pattern), postconditions, termination conditions, failure conditions and the execution part of the PoB. This information is used at runtime by the BBC to check if the current executing PoB is still valid.

The Plan-based Component is the third element of the control unit. It contains a planning mechanism capable to deliberate about single agent plans. The plans are hierarchical skeletal plans whose nodes could be new subplans, executable PoB or primitive actions. The plans are stored in the Local Plans Knowledge Base. The component shall be able to devise and monitor its correct execution a plan when requested, devise but not execute a plan, evaluate a plan and interpret a plan received from another agent.

The CC contains a mechanism for devising joint plans. It has access to protocols, a library of joint plans and knowledge about communication strategies stored in the cooperation knowledge level of the KB. It is important to notice that this layer is used to generate collaboration among agent in a Multi Agent System, which is modelled and integrated in the agent model itself.

The access to the information is hierarchical and must be passed among levels. The idea is that lower level information is visible from higher level layers, but not vice versa. The flow of control between layers are performed by means of messages. A layer can communicate with adjacent layers. An important field of the message it the type data. There is a fixed type of message that are allowed to pass between different layers.

A simulation example is provided with InteRRaP. The scenario is a loading-dock with shelves with different goods, a loading truck with goods and different robots that load and unload trucks. The robots must also negotiate their movement with

other agents when a bottleneck is detected, there procedures are stored in the CC component.

TuringMachines

TuringMachines was introduced in [36] by Innes A. Ferguson and it is a 3-layered horizontal hybrid architecture. The architecture layout is shown in figure 28.

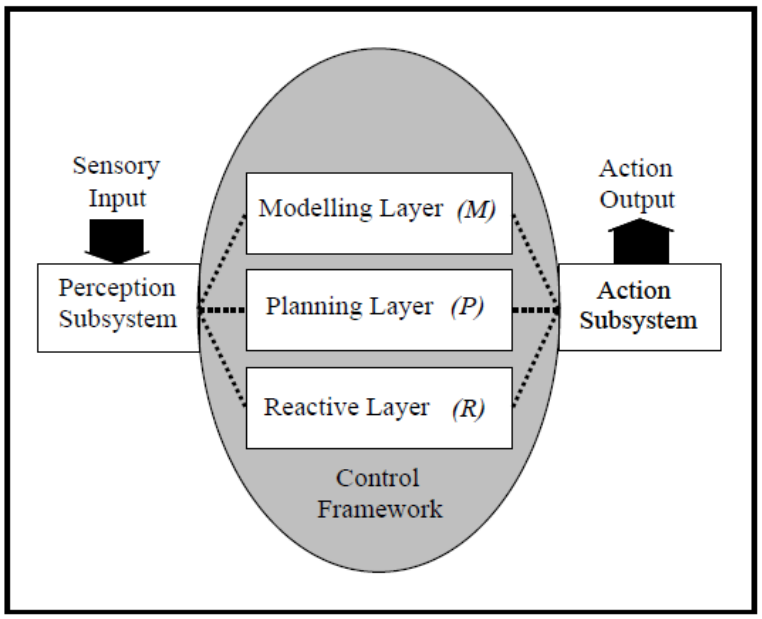

FIGURE 28 : TURINGMACHINES ARCHITECTURE FROM [36]

It is composed by three separate control layers: a reactive layer *R*, a planning layer *P*, and a modelling layer *M*. As it is possible to notice, the three layers are directly connected to the sensory layer (through the Perception Subsystem) that uses an internal computational mechanism to processing appropriate aspects of the received information, and to the effector apparatus (through the Action Subsystem) to which they send appropriate motor-control and communicative action commands.

The three layers operate concurrently, they are independently-motivated, activity-producing and mediated by a control framework. Each layer is designed to model the agent's world at a different level of spatial-temporal abstraction and so is endowed with different task-oriented capabilities. The layer R is capable to provide fast reactive capabilities to cope with immediate short-term events, while layer P can generate long-term plans and finally layer M can incrementally store and

provide mental and partial model of the world entities. Collectively, the three control layers are aimed at providing a TuringMachines agent with a variety of deliberative and non-deliberative task-achieving behaviours; these include behaviours that are situationally determined or reactive, goal-directed, reflective and predictive.

The Perception and Action Subsystems provides for sensing and acting in a dynamically changing world. Input and output are synchronized and the processing cycle is managed by an internal agent clock. Each synchronization is called time-slice. The perception system is composed by two components: a set of symbolic sensors and a perception buffer, the first can cope with the real hardware that generate the data, the second acts as a temporary buffer where the information is stored and acquired by the three layers in a multi-attribute record. The action system, in a similar fashion, is composed by a limited-size action buffer for receiving the motor-control and communicative action commands sent by the agent's control layers and a series of effector capable to translating these commands in a concrete action in the real world.

The Reactive Layer R is endowed with the capability to react fast and robustly to externa unpredicted events. Its main component is a series of situation-action rules. There is no model of the world in this layer, only actual world assumptions are taken in account. A situation action-rule consists of two parts: a condition set and an action. The condition set defines which conditions must met, the action is the reactive command for the agent's effector. Because rules operate in parallel, more of them could try to send different command to the same effector, generating a conflict. In order to cope with this situation a filtering mechanism selects one of the actions. Moreover, when a new rule is triggered, a message is sent to layer M.

The Planning Layer P has the duty to build and execute plans with the aim to achieve the agent's tasks and goals. The functionality of this layer is distributed in two components: the Focus of Attention Mechanism and the Planner. The first acts as a filtering module, selecting the most relevant information and events in the world to pass to the planner. The second module generates effective goal-oriented behaviour, instead of a fully plan (like the STRIPS planner).

Finally, the Model Layer M has the role to provide reflective and predictive capabilities to the agent, allowing it to interact in a multi-agent environment. It provides to the agent tools to maintain mental and causal models of the world. This layer is divided in three components, a focus attention mechanism (like the P layer one) an Explanation module and a Prediction module. The Explanation module is capable to provide plausible or inferred reasons about the behaviour that has been observed, by building and maintaining models of all observed entities and by detecting discrepancies between entities current behaviour and the anticipated one. If a conflict is detected, this module tries to explain why the conflict has been generated. The predictions of the behaviour are provided by Prediction module that builds simulations of each entity that is modelled. This module has also the role to provide solutions to resolve detected conflicts. Finally, this module is able to construct expectations to be used subsequently by the Explanation module.

A consequence of using three concurrently-operating activity-producing layers, is that they have independent access to sensors and effectors of the agent, allowing conflicting access to them. Two types of conflicts are taken in account: different actions from different layers to manage the same event in the environment and different action for different goals. These conflicts are managed by mediatory control rules. These rules are applied at the beginning and at the end of a time-slice. If the rule is applied at the beginning of the slice it is called censor rule because it filters some information to some layer, if it is at the end of a time slice, it is called suppression rule, because It silences some action command from some layer to the action system.

2.2.3.5 Cognitive Architectures

Cognitive architectures tackle the problem of decision making by taking reference from the human mind. In these architectures both common human processes (as example memory, learning) and apparatus (vision, different components of the brain) are represented in some sort of software program. Research related to cognitive architecture can be subdivided in three approaches [37] [38] , that represent cognition in different ways: *Symbolic* (or cognitivist), *Connectivist* (or emergent) or *Hybrid*.

The symbolic or cognitivist approach, that is the predominant one, represent cognition by means of symbolic representation. All the processes related to perception, reasoning, learning, adaptation, memory storing and action are manipulation of these symbols in appropriate data-structures. This approach has also been labelled as the information processing approach to cognition. Because the symbolic representations are the descriptive product of a human designer, they can be accessed directly and understood or interpreted easily by humans. This is a great advantage but has also drawbacks: the representations are dependent by the programmer and they constraint the system to an idealized description of the processes of the human activity. This approach leads to usually complex architectures, limited by the symbol representation. It is possible, however, to extend the available knowledge by means of machine learning algorithms, probabilistic modelling or other techniques to deal with the uncertain, time-varying and incomplete nature of sensor data used. This type of approach generates usually models tailored for well-defined problem domains, while showing difficulty to handle complex, noisy and dynamic environments. It is also very difficult to gather higher order capabilities such as creativity or learning.

The connectivist or emergent approach, cognition is the process of adapting to the environment. The goal of this kind of system is to maintain its own autonomy by means of self-organization processes through which it reacts on the environment to maintain a real-time operability. Perception is a fundamental phase of this approach because is the acquisition of sensory data, at the contrary in a cognitivist approach the environment and the perception itself is modelled. This cognitive agent constructs its internally represented reality as a result of its operations and experiences from the world. Connectivist systems have also the possibility to get familiar with and learn how to control the body it is embodied in. This allows to the designer to do not model each body-characteristic into the system. Although very powerful, these systems are very complex and hard to model.

The hybrid approach combines aspects of the symbolic and connectivist to represent cognition. The idea is to avoid explicit programmer-based knowledge in the creation of artificially intelligent systems and to use perception-action behaviours rather than the perceptual abstraction of representations. These

systems are able to exploit symbolic knowledge to represent the agent's world and logical role-based systems in order to reason about its knowledge in order to expose a goal-oriented behaviour. In the same time, they use an emergent model of perception and action able to explore the world and construct its knowledge.

Two cognitive architecture are proposed: SOAR as example of a cognitivist architecture, and CLARION as an example of a hybrid one. A review of more different architectures based on different approaches can be found in [37] [39]and in the references therein.

SOAR

Soar (State, Operator and Result) [40] has its root in the classical AI and based on Newell's physical symbolic hypothesis [41] and is considered one of the first cognitive architecture proposed and it is based on the cognitivist approach. Its main goal is to handle the full range of capabilities of an intelligent agent through a general mechanism of learning from experience.

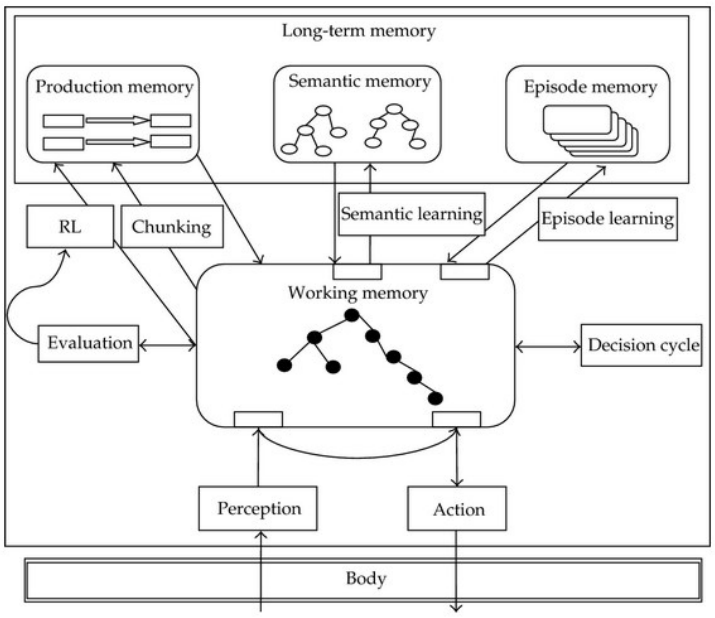

FIGURE 29 : SOAR ARCHITECTURE FROM [40]

The architecture of SOAR is shown in figure 29. It is possible to notice different structures related to memory such as long-term memory (to store knowledge in general), split in Procedural (to store knowledge of performing tasks), Semantic (to store declarative knowledge related to facts of the actual world) and Episodic (to

store knowledge about past events experienced). The first two are universally applicable, the last one is considered context-dependent. Furthermore, SOAR tries to solve problem by means of the only Procedural memory. If this type of memory is not enough, Semantic and Episodic are employed to aid in problem solving. Procedural memory stores SOAR's knowledge of how to select and perform discrete actions, in a form if if-then rules called productions. Production fire in parallel whenever they match working memory. Productions allows the proposition, evaluation, selection and application of operators. Operators are the main feature that allows decision making in SOAR. Operators are proposed (created in working memory) by rules of the procedural memory based on the current context. Additional rules can evaluate the proposed operators, creating preferences among them.

The working memory stores all knowledge that is relevant in the current situation. It contains the goals, perceptions, hierarchy of states, and operators. The goals direct the architecture in the desired state and states give information about the current.

The *Decision Cycle* is the process to support cognition. Its elaboration its composed by two phases: elaboration and decision. During the first phase has parallel access to Long Term Memory to update features and values stored in the working memory. In the second phase, an operator is chosen in the current problem space and applied, in order to reach the goal. When the choice of a unique operator to perform is not unambiguous, SOAR reaches an impasse. When this happens, SOAR sets up a new state in a new problem space, with the goal of resolving the impasse: this is the way in SOAR to implement learning, by actively reasoning how to solve the impasse. This process is called universal subgoaling. Three causes of impasses are considered:

- No operator is proposed: need to fine an applicable operator
- Two operators are proposed and SOAR is unable to define which one is ore applicable: need to choose between them
- An operator is proposed, but it doesn't know which chance will happen if it uses it: need to find a state that implements the operator

When the impasse is solved, a new production summarizing the processes that occurred in the substate in solving the subgoal is created (this operation is called

chunking) and stored in the Procedural memory. If SOAR will face the same or analogous problem, it will be able to exploit the new production.

CLARION

Connectionist Learning with Adaptive Rule Induction On-line (CLARION) [42] is considered a hybrid architecture because it combines connectionist and symbolic representation and combines implicit and explicit psychological processes. Overall CLARION is a modular architecture composed by different functional subsystems (with many modules within them). It has a dual representation structure, with both implicit and explicit representations in separate modules within each subsystem. Implicit processes are generally less accessible and more "holistic" while explicit processes are more accessible and crisper. The schema of the architecture is shown in figure 30.

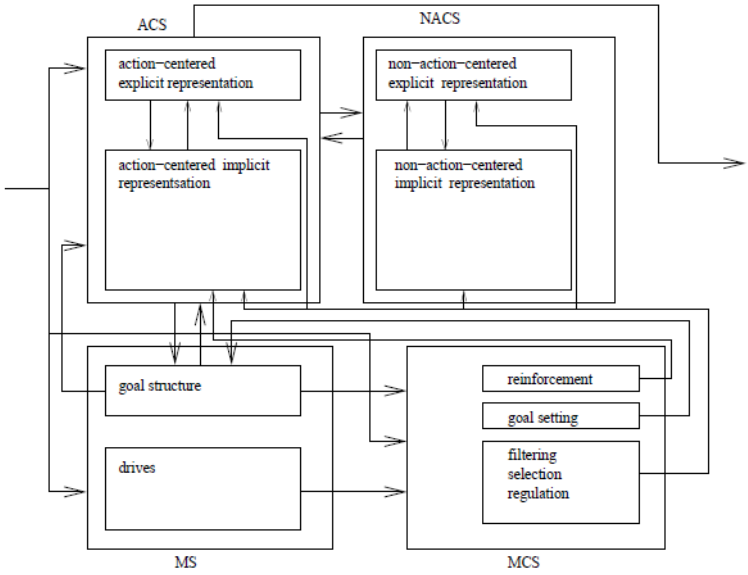

FIGURE 30 : CLARION ARCHITECTURE FROM [42]

The subsystems include the action-centered subsystem (ACS), the non-action-centered subsystem (NACS), the motivational subsystem (MS), and the metacognitive subsystem (MCS). Each subsystem has two different modules with different levels of representation. The top level encodes explicit knowledge while the other encodes implicit memory. Together they form a dual representation structure.

The role of the ACS is to control action, based on procedural knowledge, regardless of the action is an actuation in the physical environment or internal mental operations. The implicit layer is formed by Action Neural Network, while the explicit one is composed by action rules. The NACS maintains general knowledge for retrieval of appropriate information and inferences on that basis (ultimately in the service of action decision making by the ACS). The knowledge store can be semantic (general knowledge) or episodic (knowledge of past events). The MS provides underlying motivation for perception, action and cognition and, finally, the MCS monitors, directs and modifies the operation of the other subsystem in order to improve the overall performance of the system.

2.3 Multi Agent Systems

Since their beginning during '80s Multi Agent Systems(MAS) have been considered as an entity, composed by a society of agent, that interact together in order to coordinate their behaviour. It was obvious that the environment where agents need to operate (cooperating and collaborating) are very complex, and a modular, systematic and distributed approach was necessary to manage this overall complexity

Langley stated [43] that there are three different approaches to carry out these problematics:

- application of principles and techniques from software engineering, which are used regularly in developing traditional largescale software systems;
- Cognitive architecture;
- Multi agent systems.

By another point of view in [44], because there are problems related to restriction to computational and knowledge limitations, no unique agent can manage the succeed in its task and cooperation become mandatory. In these conditions two concept could be useful to overcome these difficulties: modularity and abstraction. A MAS offers modularity. If a problem is rather complex the only way it can be reasonably addressed is to develop a number of functionally specific and modular

components called agents, which are specialized to solve a particular problem aspect.

The most important characteristics of MAS are:

- **Autonomy**: the agents have at least a minimum grade of autonomy
- **Local views**: no agent as a global knowledge of the environment or a single software could not be able to process all the information from the environment
- **Decentralization**: there is no designated controlling agent
- **Asynchronous Computation**

The main motivation to develop and research MAS technology are:

- Solve problem too large for a centralized agent;
- Allow interoperation between existing legacy systems;
- Provide solutions to problems that can naturally be regarded as a society of autonomous components that are able to interact;
- Provide solutions that efficiently use information sources that are distributed;
- Provide solutions that efficiently use distributed expertise;
- Enhance performance along the dimensions of: computational efficiency, reliability, extensibility, robustness, maintainability, responsiveness, flexibility and reuse.

First of all, a characterization of different typologies of approaches to the MAS theory is proposed from the actual state of the art, then three notable examples of MAS are exposed then some real-world applications and concrete MAS framework are introduced.

2.3.1 MAS characterization

Aim of this paragraph is recall the most used MAS characterization. In order to analyse a MAS architecture in [45] and in the references therein two dimensions are proposed: the *awareness/unawareness* of agents of the existence of the organization structure and the *type of architecture*. The type of architecture can follow two

different approaches, which take in account if organization is considered a process inside the MAS or an entity by itself.

In the first case organization is considered a process endowed with the task to organize a set of individual agents and it is linked to the *Agent-Centered MAS (ACMAS)* approach. In this approach, the focus is given to individual agents, to their local behaviour and interactions, without concerning the global structure of the system. The main idea is that organization is a phenomenon that emerges thanks to the collective behaviour of different agents that have individual behaviours, total control on their actions and interact in a common shared and dynamic environment. This global process is usually called self-organization. An example of this approach is ant colony, where there is no organizational behaviour and constraints are explicitly and directly defined inside the ants. In [46] some problem with ACMAS are appointed such as interaction pattern hard to track and the difficult to predict the global behaviour caused by the strong possibility of unwanted emergent behaviours.

The second approach, called *Organization-Centered MAS (OCMAS)* considers agents' organization as an entity with its own requirements, its own objectives and is represented by a group of agents. The use of an organization provides a new way for describing the structure and the interaction that take place in a MAS, allowing to decrease complexity in the development of the single agents while increasing efficiency and the capability to model the problem to solve. This organization model that represents the MAS is an abstract representation of the concrete organization. The model describes the expected relationships and patterns of activity which should occur at the agent level and therefore the constraints and potentialities that constitute the horizon in which agents behave.

It is possible to recognize two different levels of the organization: the organizational structure and the concrete organization. The first is what persists when agents enter or leave the organization and it is the static aspect of the organization, the second, which resides at agent level, is one possible instantiation of an organizational structure and is related to the dynamic aspect of the organization, defining also rules to join and leave organization.

Some features are essential in a OCMAS model:

- An organization is constituted of agents (individuals) that manifest a behaviour.
- The overall organization may be partitioned into groups(partition) that may overlap.
- Behaviours of agents are functionally related to the overall organization activity (concept of role).
- Agents are engaged into dynamic relationship which may be "typed" using a taxonomy of roles, tasks or protocols, thus describing a kind of supra-individuality.
- Types of behaviours are related through relationships between roles, tasks and protocols.

There are some principles that must be followed [46] to develop a mas following an OCMAS approach:

Principle 1: the organization level describes the "*what*" and not the "*how*". It defines and imposes a structure into the pattern of agents' activities, but does not define how agents behave, it has not any code to be executed by the agent, but provides specifications, in form of norms or laws, of the limits and expectations that are placed on the agents' behaviour.

Principle 2: no agent description and no mental issue at this level. The organizational level should not say anything about the agent would interpret this level. Any kind of agent (from reactive through cognitive) may act in the organization. Moreover, this level should get rid of any mental issue such as beliefs, desires, intentions, goals etc.: it should provide only description of the expected behaviours.

Principle 3: an organization provides a way to partitioning a system, each partition constitutes a context of interaction for agents. Thus, a group is an organizational unit in which all members are able to interact freely. Agents belonging to a group may talk to one another, using the same language. Whereas the structure of a group A may be known by all agents belonging to A, it is hidden to all agents that do not belong to A. Thus, groups are opaque to each other and do not assume a general standardization of agent interaction and architecture.

Four different models of MAS architectures are addressed in [45], by arranging the two dimensions (Architecture and awareness) in a Cartesian space as shown in figure 31:

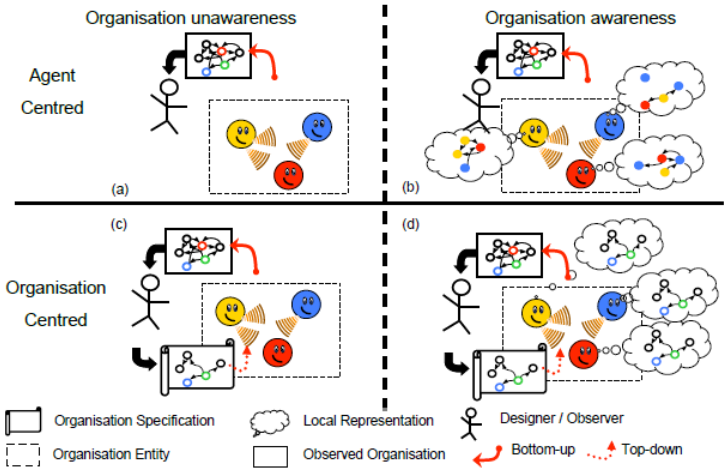

FIGURE 31 : MAS ARCHITECTURE MODELS FROM [45]

- **Emergent Organization MAS (Agent Centered/Unaware)**: the agents don't represent the organization (since it is not modelled), although the external observer can see an emergent organization. Agents are unaware that they are part of any kind of organization
- **Coalition Oriented MAS (Agent Centered/Aware)**: each agent has an internal and local representation of cooperation patterns which it follows when deciding what to do, this local representation is obtained either by perception, communication or explicit reasoning.
- **Agent-oriented software engineering (Organization Centered/Unaware)**: the organization exists as a specified and formalized schema, made by a designer but agents don't know anything about it and even do not reason about it. They simply comply with it as if the organizational constraints were hard-coded inside them.
- **Organization Oriented MAS (Organization Centered/Aware)**: agents have an explicit representation of the organization which has been defined, the agents are able to reason about it and to use it in order to initiate cooperation with other agents in the system.

Three MAS models will be exposed. The first two of them (AGR and MOISE) are aware OCMAS approaches, where the agent are aware of the presence of other agents in the system, the third one (MACODO) is a hybrid approach: there is a defined model and structuration of roles in the agency, but the agents are free to self-organize if necessary.

2.3.2 Reference Models

2.3.2.1 AGR

The AGR (Agent-Group-Role) model, also known as Aalaadin model [47], is a generic meta-model for a MAS, and it is based on three core concepts: *Agent*, *Group* and *Role*. The disposition of these concept is shown in figure 32.

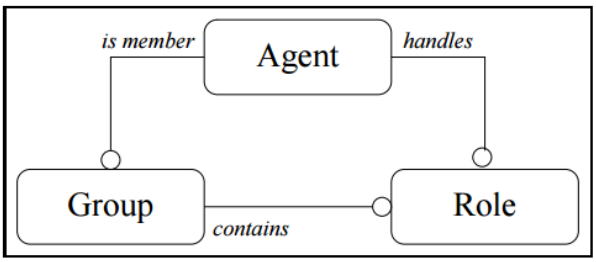

FIGURE 32: THE AGR MODEL FROM [47]

The *Agent* is specified as an entity that can play a Role within Groups. The model doesn't apply any constraint to how the agent is developed or modelled.

The *Group* is considered as an aggregation unit. Basically, it could be a used to tag a set of agents. It is defined by the tuple:

$$< R; G; L >$$

R is a list of roles, G is an oriented graph specifying valid interactions between two roles and L is the interaction language.

Groups have these characteristics:

- An agent can take part to more than one group.
- Can overlap
- Can be founded by any agent, and other agent must ask for admission
- Can be distributed among different machines.

The *Role*, finally, is an abstract representation of an agent function, service or identification within a group. Each agent can handle several roles, and each role handled by an agent is local to a group. As with group admission, handling a role in a group must be requested by the candidate agent, and is not necessary awarded. By relating communications to roles, and by authorizing an agent to play several roles, our model allows agents to handle several heterogeneous dialog situations simultaneously.

A role is characterized by:

- **Uniqueness**: a role can be unique or multiple within a group. A role identified as unique must be held by one only agent within a given group.
- **Competence**: identifies a condition an agent must satisfy to be able to play the role within the group.
- **Capacity**: a property awarded to an agent when it plays this particular role.

To demonstrate the effectiveness of the model, a framework called MADKIT (Multi Agent Development Kit) had been developed [48]. It is able to manage the agent-group-role model and has three design principles: micro-kernel architecture, agentification of services and a graphic component model. The micro-kernel can manage agent life-cycle, control of local groups and local message passing management.

2.3.2.2 MOISE

Moise [49] extends some aspects of AGR model, by structuring its proposed model in three levels:

- **Individual level**: for each agent, there is the definition of the task that it is responsible for, based on the concept of *Role*, that is able to constraint the agent's individual behaviour;
- **Aggregate level**: takes in account aggregation of agent in large structures, based on the concept of *Group* that can constraint the layout of agents that take part are part of strong interactions between each other's;
- **Society level**: in this level, the global structure and interconnection of the agent inside the agency is defined and it is based on the concept of

Organizational Link, that can regulate social exchanges between the agent society;

It defines three different aspects of the organization, gathered in a unique Organizational Specification (OS), a web of roles, groups, and links which is a way for the designer to structure the system independently of the agents in the system. The Organization Entity (OE) is considered a set of agents functioning under an OS, and it is essentially a real-world instance of the OS. Each aspect covers a different level of the organizational model.

The first aspect of the OS is *structural specification* (SS). This specification has a similar aim to the AGR model, defines the roles and groups inside the organization. It is defined by a tuple with the following fields:

<R; C; rg; L>

R is set of identifiers of roles, *C* an inheritance relation among roles, *rg* the root group (the groups are specified by a Group Specification GS) and *L* a list of links between roles. A *link* is composed by a tuple with 4 fields: source

<s; t; k; p>

Where *s* is the source role, *t* is the target role of the link, *k* is the type of link and *p* is the scope of the link. Three types are defined:

- **Acquaintance**: the link allows to an agent playing a role to acquire information about another role of another agent;
- **Communication**: the link allows to an agent playing a role to communicate with another agent with a precise role;
- **Authority**: the link defines a power relation between two roles;

While two scopes are defined: intra and inter. This parameter specifies if the link is intra or inter groups.

A *group specification* is defined by the following elements:

<id; compat; maxrp; minrp; maxsg; minsg>

Where, in order, it defines a unique ID for the group, a map of roles, a maximum and minimum cardinality for each role and a maximum and minimum cardinality of subgroups.

Functional specification (FS): defines the mission that the agent in the MAS has to achieve. It defines a tuple with three fields:

$$<M; G; S>$$

Where *M* is a set of mission identifiers, *G* is a set of goals of the organization and *S* a set of scheme specification. The goal is specified by an identifier, a list of mission containing the goal, a type parameter that can be achievement or maintenance, a cardinality of agent that must fulfil the goal in order to consider it achieved a deadline and a plan, that is a list of sub-goal that decompose the goal. The scheme specification defines schemas for goal decomposition.

Finally, the Normative specification states both the required roles for missions and missions' obligations for roles. It is composed by a list of norms. A norm is defined by an id, an activation condition, a role, a type parameter that can be obligation of permission, a mission and a deadline to fulfil the mission.

2.3.2.3 MACODO

MACODO [50] (Middleware Architecture for COntext-Driven dynamic agent Organizations) has two different objectives:

- Develop an organization model able to formally define abstraction that allow to define dynamic agent organization;
- Describe the components of a distributed middleware capable to handle the model and the agent life cycle.

The model is formally defined by means of the Z language [51]: it defines the core data models as group and role and functions and operation schemas to describe laws that represent the behaviour in the MACODO framework. A synthetic schema of the model is shown in figure 33.

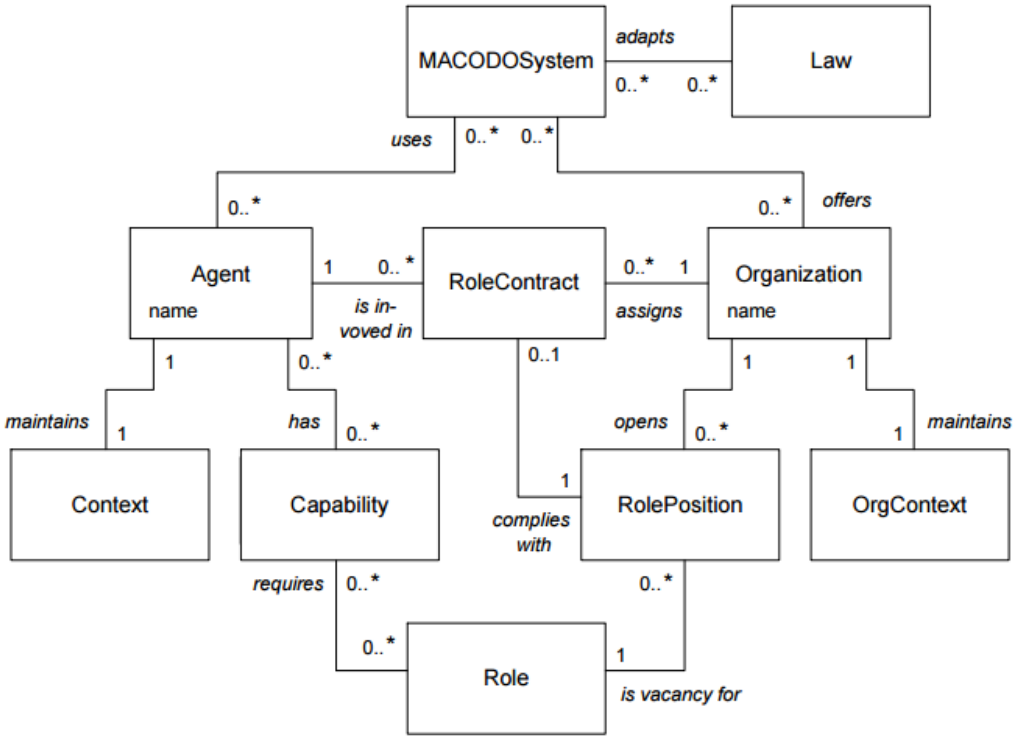

FIGURE 33 : MACODO ORGANIZATION MODEL FROM [50]

The main concept in MACODO are:

- **Context**: represents information in the environment of an agent that is relevant for the organizations in which the agent participates
- **Capability**: refers to ability of an agent to perform tasks. Capabilities describe required qualifications of an agent to participate in an organization. How an agent uses its capabilities to achieve its goals is an issue private to the agent.
- **Role**: A role describes a coherent set of capabilities that are required to realize a functionality that is useful in an organization. It is defined as a set of capabilities
- **RolePosition**: A role position is a vacancy for a specific role in a specific organization.
- **RoleContract**: is an agreement between an agent an organization that allows to the agent to play a specific role in a specific organization.

- **Agent**: an agent is defined as an entity able to take a role inside the MACODO system. It is defined by a unique name, a set of capabilities, a context and a list of role contracts. An agent can only play roles for which all the required capabilities are fulfilled. An Agent must have at least a capability.
- **OrgContext**: Organization context represents all the relevant information upon which the dynamics of the organization depend. Useful for organizations dynamics.
- **Organisation**: Is a container that allows the agent to collaborate. It is defined by a name, a set of RolePosition, a set of RoleContracts and by a context.
- **Law**: they define how agent can perform specific action in the MAS such as create, join or leave a group,
- **MACODOSystem**: the whole system is defined as a set of organizations and a set of agents.

The middleware has the role to handle role positions and manage role contracts, to maintain the organization context in a distributed environment and to enforce laws. Other requirements are: adaptability, robustness to node failure, scalability and portability.

The middleware is arranged in a four-tiered architecture as shown in figure 34:

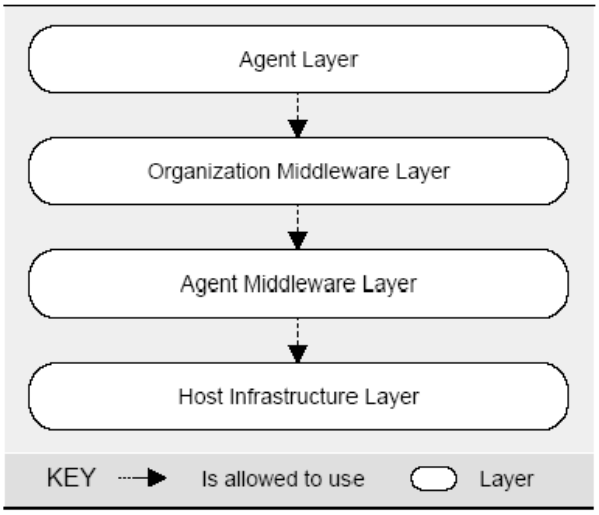

FIGURE 34 : MACODO MIDDLEWARE ARCHITECTURE FROM [50]

- **Host Infrastructure Layer**. The host infrastructure encapsulates common middleware services and basic support for distribution, hiding the complexity of the underlying hardware.
- **Agent Middleware Layer**. The agent middleware layer provides basic services in multiagent systems. It includes basic support for perception, action, and communication.
- **Organization Middleware Layer**. The organization middleware layer provides support for dynamic organizations. This layer encapsulates the management of dynamic evolution of organizations and it provides role-specific services to the agents for perception, action, and communication. The organization middleware layer is the main focus of the software architecture described in this section.
- **Agent Layer**. The agents in the agent layer use the organization middleware to interact with the environment and each other through the roles they play in the organizations.

The middleware and the organizational model has been used to develop a prototype platform for a traffic monitoring application to assess its adaptability, scalability and robustness.

2.3.3 MAS applications in different scenarios

The MAS theory proved to be useful is different kind of situations and scenario, as a general tool to design and simulate coordination and behaviour of different interoperating entities.

Examples of employment of the MAS theory can be found in the energy scenario [52] and [53] with specific applications related to power system restoration [54] and protection [55] with a particular effort focused to the development of solutions tailored to microgrids ([56] and [57]) that can exploit the distributed nature of Multi-Agent Systems to obtain benefits such as increased autonomy, reactivity, proactivity and social ability. In [58], a review of solutions employed to manage microgrids, MAS are in fact declared the main solution in the decentralized approach for the so called “secondary control”, which is endowed with the task to allow economical and reliable operation of the microgrid. More specifically in [59], it is

shown that MAS are exploited to develop tailored controllers are designed to manage optimization and power restoration in microgrids. Another employment is in the context of the industry and industrial production, where MAS are employed mainly for manufacture process managing ([60], [61] and [62]) and assembly. In [63], for example, a multi-agent model is employed to self-organize the behaviour of two or more robotic system to allow collaborative operation in the same assembly workspace.

Furthermore, a wide employment of the MAS theory is related to automation. It is possible to detect at least two important applications: home automation and robotics. In the first scenario MAS has been extensively used to provide intelligent to home apparel in order to distribute and manage different resources. For example, in [64] authors exploited a MAS with a proper model to define and manage a Home Automation System (HAS), in [65] a BACnet intelligent home supervisor system was implemented by means of BDI agents while in [66], [67] and [68] the focus is in the managing and orchestration of the requirement of energy in smart homes by using a multi-agent approach.

In the second scenario MAS are extensively used to coordinate the behaviour of different components or robots to reach a common goal, which is not satisfiable or less performing if carried out by a single entity. Some common tasks that can be found in the state of the art are, for example, navigation([69], [70] and [71]), localization([72], [73] and [74]), exploration and mapping([75], [76] and [77]), search and rescue([78] and [79]) and human assistance([80] and [81]). A particular interest in in the coordinate movement of robots by means of a flocking behaviour.

Other applications are related to some specific cooperative tasks such as foraging, where agents are endowed with the task to collect resources using different approaches such as stigmetry [82] to avoid communication bottleneck. In this scenario agents were able to coordinate to find a target without direct communication. Another example is [83], where authors exploited Augmented Markov Models as a tool to define the behavioural dynamics of different robots. Other tasks are object moving and pushing and toxic waste cleaning. In the first one agents are used, for example, to co-ordinately move furniture [84] by means of a

global control able to co-ordinately change the pose objects. Another similar example is shown in [85] where, more in detail, the procedure to coordinate the actions between agents are explained.

More specific employments exploit MAS to organize the entities of soccer teams of robots [86] and [87]. This scenario is considered an excellent platform for testing artificial intelligence platforms in an adversarial environment, and competitions are frequent (such as FIRA or RoboCup).

One environment that profited from the use of this theory is the marine one. In fact, numerous studies were performed to design MAS tailored for underwater and surface vehicles that collaborate usually with a ground station to cope with this difficult environment. In fact, these systems must take in account many issues such as degraded visibility, slow communication and a highly variable environment. Some studies are focused in specific tasks such as mine countermeasures and marine surveillance [88].

Moreover, particular algorithms are developed to manage different controls extended to be used in a multi-agent environment such as fault detection [89] and robot formation([90], [91], [92] and [93]).

In addition, MAS are extensively used as a tool to provide simulation in different scenario such as the economical one, where these technologies are employed to simulate markets where a focus is dedicated to the simulation of the energy market ([94], [95], [96] and [97]). Simulations with MAS are exploited to simulate crowd behaviour, where each agent acts autonomously in different environments [98] and situations, such as an emergency to test evacuation plans [99]. Another application is related to medicine [100] where the MAS theory is applied in simulations related to nano-robots drug delivery in cancer body tissues

An important orthogonal field of study is the development of different, general purpose framework to design and develop MASs (like MACODO shown in the previous section) implementing common standards to save developers time and aid the standardization of MAS development. Before introducing other reference framework, it is important to introduce FIPA (Foundation of Intelligent Physical Agents) [101], one of the most widely used and accepted standards and

specifications to promote agent-based technologies and interoperability of its standards to other technologies. The specifications regard different aspects of MASs such as agent communication, agent transport, agent management, abstract architecture and applications.

JADE [102] is a Java based multi-agent system and run-time middleware to develop application that are compliant to FIPA standards. JADE most notable features are transparent message-passing layer based on FIPA ACL (agent communication language), white and yellow pages services, support for strong mobility, and built-in FIPA protocols. An agent is defined as a Java Thread Object with a set Behaviour attached. These behaviours represent task-achieving jobs, and they are executed by a round-robin scheduler internal to the agent. Agent can communicate between each other by means of FIPA ACL messages.

Jason [103] is, at the same time, an agent language, an agent development framework and a run-time system. As a language, it implements a dialect of AgentSpeak [104] that is a language based on logic programming and the BDI architecture (Belief-Desire-Intention, explained in 2.2.3.3) for develop cognitive autonomous agents. As a development framework, API are provided to design agents and MAS. As running system, it provides a centralized infrastructure and a set of tools in Java to execute the multi-agent system, but the system is easily extendible and a proof of this is that it is possible to use JADE as agent container for Jason agents with a proper wrapper to adapt Jason agents to the JADE ones.

An agent in Jason is defined by the following architecture, which is executed repeatedly by the agent container:

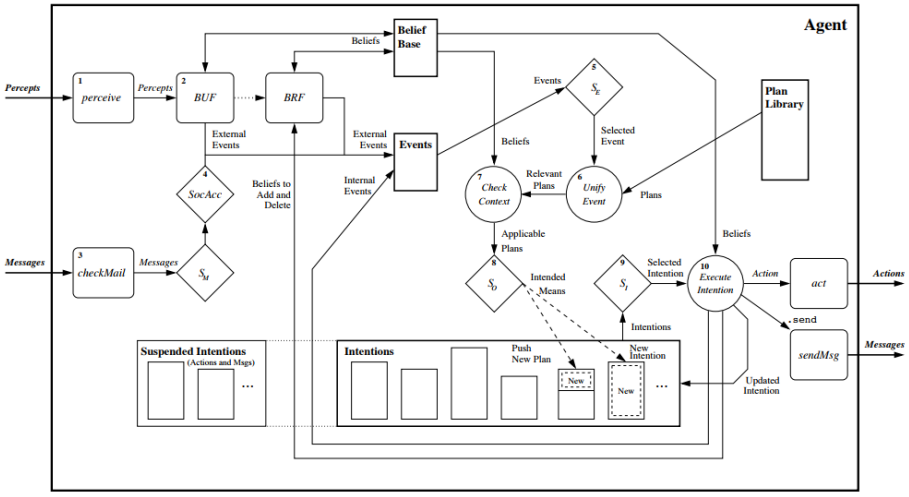

FIGURE 35 : JASON AGENT ARCHITECTURE FROM [103]

It is important to notice that this kind of agent has a concept of Environment embedded, in fact, there are modules tailored to perceive and act.

The CArtAgO (Common ARTifact infrastructure for AGents Open environments) [105] framework and infrastructure is based on the A&A(Agents&Artifact) meta-model [106]. Artefacts are tools that the agent can use to enhance their capabilities for achieving their own goals, in a similar way human do with their tools. Artefacts are used by the agent as an environmental resource to act or sense. They include different technologies such as sensor, actuators, databases etc. These artefact-based environments are structured in open workspaces, possibly shared between different nodes of the MAS network.

2.4 Environment

The environment is a fundamental aspect of the life cycle of robots performing a mission. As per definition, a Robot is able to perceive the environment and to act in it through its sensing and acting RobotParts, while the knowledge of presence of other robots or a model of the surrounding area is fundamental to reason and coordinate to fulfil a mission. So, it is important to characterize it and define roles of the environment in a Multi-Agent System.

2.4.1 Environment Characterization

2.4.1.1 Environment Dimensions

In [10] the authors provide some dimensions to determine the task environment. These dimensions should be taken in account during the design of the model of the environment and of the agents that live in it. These dimensions are:

Fully Observable or Partially Observability: if an agent's sensors give access to all the relevant data at each point of time, the task environment is Fully Observable. This typology of environment is convenient because, thanks to the fact that the agent is able to percept any useful information from the environment, it is not necessary to store an internal state to keep track of the world. An environment might be Partially Observable if the sensor used to percept are noisy or because some part of the state is missing from the sensor data. For example, in a search mission and a fully observable environment, the robot would be immediately able to know the position of the target to be found. If the agent has no sensor, the environment is unobservable.

Single agent or Multiagent: meanwhile the distinction between single agent and multiagent environment is quite straightforward, it is necessary to define different multiagent environment. Multiagent environment can be divided in competitive and cooperative. Obviously a multiagent environment has more challenges to be addressed such as communication, cooperation and coordination between agents.

Deterministic or Stochastic: if the next state of an environment is completely determined by the current state and the action executed by the agent, the environment is considered Deterministic. By the other way, it is considered Stochastic. An environment is considered *Uncertain* if it is not fully observable and deterministic.

Episodic or Sequential: if an environment is episodic, the agent's experience can be divided is single atomic episodes with a proper cycle percept-think-act and each decision is not linked to previous decisions. In a sequential environment, at the opposite, the current decision could affect future decisions, so the agent needs to think ahead in order to choose its action.

Static or Dynamic: in a Static environment, while the agent is deliberating, no change could happen. At the opposite, the environment is Dynamic. If the environment itself doesn't change, but the agent performance score does (for example, if too much time is spent on reasoning), the environment is considered *Semidynamic*.

Discrete or Continuous: this distinction is based on three factors: the state of the environment, how time is handled and to the perceptions and actions of the agent. A Discrete environment has fixed locations or time intervals or a finite set of actions that can be performed in it, a Continuous environment could be measured quantitively to any level of precision and they are based usually on fast-changing and unknown data sources.

Known or Unknown: This is linked to the knowledge of the agent about the laws that shape the environment. For example, in a Known environment, the outcome (or its probability of the environment is Stochastic) are given, by the other way, in an Unknown environment, the agent will have to adapt in order to learn how the environment works in order to make good decisions.

Real world scenarios are, usually, Partially Observable, Multi Agent, Stochastic, Sequential, Dynamic, Continuous and (usually) partially unknown. This is the most difficult combination of dimensions.

2.4.1.2 Environment representation

In [10] are presented three ways to represent concretely the environment the agent inhabits by means of software components, where the main dimensions between each one is the increasing complexity and expressiveness of the solution employed. A more expressive representation is able to capture everything a less expressive one can capture plus something more. This leads to usually more complex representations, but able to express the same concepts in a more concise manner.

These three representations are called atomic, factored and structured. Figure 36 shows these three representations.

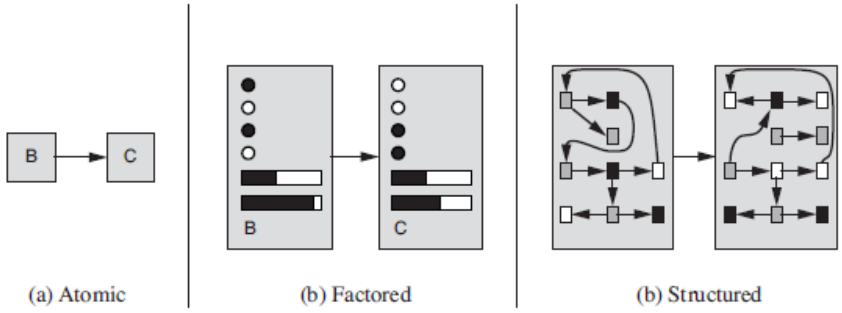

FIGURE 36 : STATE REPRESENTATIONS FROM [10]

Atomic: in this representation, each state of the world is indivisible and there is no internal structure and it is considered a single atom of knowledge, a black box whose only discernible property is that of being identical or different to another one.

Factored: In this representation, each state of the world is split in a fixed set of variables or attributes, each of which can have a value. While two atomic states have nothing (or everything) in common, two factored states could have come attributes in common while other could be different. Moreover, with factored representation, it is possible to represent *uncertainty* by leaving some attributes blank.

Structured: In this representation, the state of the world is represented by objects (that internally could have a factored or atomic representation) with explicit description of their relationship with each other.

2.4.2 Environment and MASs

In [107] an introduction of the importance of the consideration of the Environment is shown and an important focus is towards the definition of the responsibilities that the environment should implement, to be an added value to the shape of the Multi-Agent System. In fact, most approaches in agent research still views the environment as something that is modelled in the agent's minds, reflecting a minimal and implicit representation of this concept instead to being considered as a first-order abstraction. An environment designed as a first-order abstraction can be considered an independent program building block which responsibilities differs from the ones of the agents, providing a layer of abstraction working as an interface between the agents and the real world, allowing to change the module's implementation without requiring any change to other modules.

The duties of the environment are synthetized as:

- **Provide a structure**: the environment is conceptualized as a common stared space for the agents, that is used to give a structure to the whole system. The structuration could be spatial (regions, positions, locality, etc.) or organizational (the groups and roles of the agents).
- **Manage Resources and Services**: an important aspect of an organization situated in a real-world scenario is that there are usually limited resources (such as time, an instrument, the quantity of hot water in a house etc.) and a set of services that agent could need to access in a controlled way. Resources are defined as "objects with a specific state" while resources are considered "reactive entities that encapsulate functionality". The duty of the environment is to allow and monitor the access to resources and services.
- **Enable Communication**: the environment must allow the agents to communicate. This could happen, as introduced before, with two kinds of interaction: direct and indirect communication. Direct communication is mostly based on message-passing technology, in which an agent is able to send a message to another agent, while indirect communication is based on other techniques (one approach is based on stigmetry) where and agent is able to perceive and gain information from the modification of the environment around him or by communication objects produced by an agent within the environment, that other agent can detect and consume (like a simulated pheromone).
- **Rule the multiagent system**: another important aspect of the environment is the use of a set of rules or laws to manage the multi-agent system. Rules could limit the access to specific resources or services, determine the outcome of agent's interactions or monitor the consistency of the overall system.
- **Observability**: a feature of the environment is its complete observability. This allows to agents to discover and understand at run-time the environment they are discovering in a transparent way. A complex environment should also allow agent to observe the actions of other agents.

This can be done by defining an environment ontology that describes the environment, its resources and services, and possibly the regulating laws.

Chapter 3: Case studies

In this chapter, will be introduced and analysed technologies and ideas developed inside the LabMACS laboratory that influenced and aid the development of the robotic infrastructure that will be explained further in the next chapters. For each technology, there will be an introduction and a list of key features that were taken in account in further developments. These are the discussed technologies:

- **DocuScooter**: a novel instrumentation to equip different underwater scooter to ease the acquisition of heterogeneous data from the marine environment. Its structure exploits a formally defined protocol, used between the whole structure, which enables modularity and a shared representation of the connected devices. This latter feature was extended in the development of the connection between the robot and its RobotParts;
- **The Home Automation System (HAS)**: this research proposed a MAS approach for home automation, by defining a formal definition of its structure. It influenced the importance of the environment and limited resource to be shared between different agents, the idea to have explicit different agent classes, from an architectural point of view, able to communicate and coordinate in the environment, and to design, along with the MAS architecture, tailored tools to aid simulation and debug of the whole system;
- **OpenFISH:** this bio-inspired autonomous underwater vehicle is a good example of robot that could be used to test the future infrastructure: it uses a unique bus to connect all the devices (I2C) and uses, in a transparent manner, two different communication technologies (Wi-Fi and an acoustic modem). Due to hardware and software limitations, the developed software is also tailored to be used in other contexts.
- **MAS for general purpose ASV**: the developed MAS exploits ROS as communication framework to enable messages exchange between behavior -based agents addressing modularity while the exploitation of a hardware structure that divides the low-level and the high-level of the agents addresses

abstraction and easiness of usage. Some ideas (dependencies, usage of ROS as communication framework, hardware abstraction) are taken in account as features of the designed infrastructure while other aspects such as the type of organization are enhanced.

- **LabMACS Integrated System Infrastructure**: provides a wide view of the different technologies employed in LabMACS, how they are structured and which communication protocols are used in order interoperation between different software and hardware solutions. The developed system must able to transparently merge with the infrastructure in a transparent way, to increase the number of available technologies and vehicle that can be used.

3.1 DocuScooter

The Green Bubbles project (www.greenbubbles.eu), funded by the EU's H2020 Research and Innovation programme under the Marie Sklodowska-Curie grant agreement N. 643712, takes care of sustainability diving. Its main goal is to maximize the benefits of the diving activities while minimizing the negative impacts, to achieve the environmental, social and economic sustainability of the whole system.

To reach them, the development of tailored IT tools to support divers in the acquisition, collection and analysis of data from the marine environment is a fundamental aspect. The usage of these tools has been proven to ease the workload and carry out efficiently different activities regarding professional and amateur diving activities [108] [109] [110].

In the context of this project, UnivPM, with the aid of two partners of the project, designed and developed an innovative modular low-cost platform called DocuScooter [111] capable of acquiring heterogeneous data from the marine environment. After the survey, the diver can upload all the data to an external service [112] and obtain an output tailored to different typologies of end users: georeferenced data, 3D reconstructions of different qualities, files ready for 3D printing etc. The system is designed to equip different commercial underwater scooters that the diver can use to gather data from the marine environment while performing his leisure activity. Figure 37 shows its architecture.

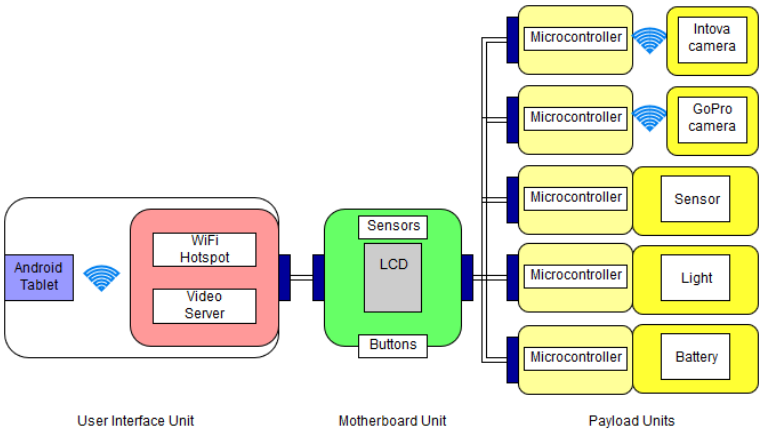

FIGURE 37 : DOCUSCOOTER ARCHITECTURE FROM [111]

DocuScooter is structured as an array of different devices called *Payload Units* composed by a commercial component (e.g. commercial action cameras, lights, sensor) and a microcontroller. All the Payload Units are connected to a main central unit, namely *Motherboard Unit*, and to an optional underwater Android tablet that communicates with the rest of the platform through a subsystem. These latter two components constitute the *User Interface Unit*.

Main ideas behind the DocuScooter platform are:

- **Modularity**: different missions require a different sensor payload so it is necessary to have a system that allows the hot plug of different components (also during the diving activity itself) without compromising the entire system.
- **Low-Cost**: to be easily employed by the diving industry, the platform must be low-cost and tailored to different available COTS (Components-Off-The-Shelf). This helps to lower the costs of the whole structure because a diver could use its own instrumentation.
- **Highly extensible**: to obtain this goal, to add a new compatible component, it is only mandatory to respect the communication protocol used to transit the data and the cable connector type.
- **Works up to 50mt**: this specification derives from the maximum depth that can be reached by waterproof housings of commercial sports cameras.

At the moment, these Payload Units are available:

- **Intova Edge-X Camera Unit, GoPro Hero 3Black Unit and Kodak PixPro SP360 Unit**: each module is composed of a microcontroller that communicates with the camera using a wireless link. Each camera has an own battery, so it does not need a power source. Exploiting the Wi-Fi communication allows all the final users to use their personal Wi-Fi camera within this infrastructure, without limiting customers only to one camera brand. In this way, the flexibility of the system is maximized.
- **Light Unit**: this unit is a single, pilotable light source of the system.
- **Battery Unit**: this unit is a single, pilotable power source of the system.
- **Sensor unit***:* it represents any specialized sensor the user wants to employ in the mission.

With the option to add as many devices as there are needs, in addition to the fact that the peripherals are independently controlled, it is possible to change each part with an updated one, for example, in the future, without compromising the entire system.

Each payload type is distinguished by means of three different values: *Type* (e.g. Camera), *Vendor* (e.g. GoPro) and *Model* (e.g. Hero3). There are two specification that allows to define the available commands for each Type or specific Payload Unit.

The first is the *Type Specification* (TS). It allows to define the available commands for each Type of Payload Unit. Each TS is a tuple composed by the Type and a list of Commands (*Generic Commands*):

<T, CMD>

The second is the *Payload Specification* (PS). It allows to define the available commands for each specific Payload Unit. Each PS is a tuple composed by the Type, Vendor, Model and a list of Commands (*Specific Commands*):

< T, V, M, CMD>

The distinction is fundamental because the protocol allows to send a command to a Payload Unit using a Specific Command (e.g. set the resolution of a camera), defined in the PS, or to all the payloads of the same type through a Generic Command (e.g. shoot a synchronized photo with all the connected cameras), defined in the TS. Specific Commands between different Payload Unit of the same Type could have

different arguments or errors, so it is necessary to make a syntactic difference to the same semantic concept.

A Command (CMD) is defined by the following tuple:

< C, REQ, RES, RETC>

The first is the *Command Code*, the second is the array or *Request Arguments* that represents the request to send to the Payload Unit (can be empty), the third is an array of *Response Arguments* that represent the reply that will be received by the Payload Unit (can be empty) and finally an array of possible *Return Code*s. Each one has its own code and explanation through a string. An *Argument* is defined by a unique name and a type. The available types for arguments are: Boolean, Integer, Float, Enum, List, String and Object.

In the DocuScooter, the tablet stores all the PSs of all available Payload Units, and has proper utilities to wrap and unwrap the messages that receives from the subsystem.

The DocuScooter could be seen as a 3-tiered architecture, which is exploited to have a high-level abstraction of the data acquired from the environment. This architecture is shown in figure 38.

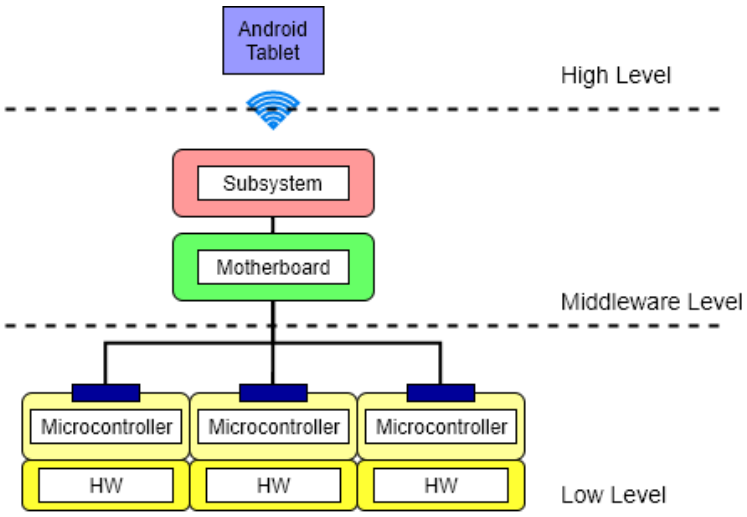

FIGURE 38 : DOCUSCOOTER 3-TIERED ARCHITECTURE

The levels are arranged in the following way: The *Low-Level* is composed by Payload Units connected in the shared bus, that are the concrete sensors (depth meter, batteries, etc.) and actuators (lights, etc.) of the system. The microcontroller applies

the first abstraction, by converting the raw data received from a sensor to the format defined in the Payload Specification and vice-versa. The higher layer can access to the information through the *Middleware Level*, composed by the Motherboard Unit and the Subsystem of the User Interface Unit. This level can monitor the status of the Payloads Units and manage the shared access to the common Low-Level bus. This level can understand if a message received from a Payload Unit is well formed or corrupted, but has no knowledge about the meaning of the messages exchanged. Through the Payload Specification, the *High Level* (in this case this component is represented by the application in the tablet) can understand which Payload are connected in the system and use them accordingly.

Extending this idea, the Middleware Level could be able to monitor the access to different Low-Level buses and devices, by providing transparency to the higher levels of the architecture. The different devices that are attached to the bus and that are used by the rest of the system can be considered *RobotParts*. For example, a depth sensor could be a RobotSensingPart with a message composed by an argument named “depth” with a value of type “float”, in a similar way a light could be a RobotActuatingPart with a message composed by an argument named “On” with a value of type “Boolean”. The idea of exploiting a common format to represent in a formal way hardware that is detected in the system is used also in the development of the infrastructure proposed in this dissertation. This allows a high grade of abstraction, by freeing the high level of the architecture to bother about how the communication with the hardware is done or to permit the same functionalities with different attached hardware.

3.2 OpenFISH

Another field of study in LabMACS, in collaboration with the DIISM department is the study, design and development of Autonomous Underwater Vehicles (AUVs). One of the output of this collaboration is the development of OpenFISH, a bio-inspired Autonomous Underwater Robot able to exploit an ostraciform swimming model to improve its performance and battery life. In LabMACS, we focused our research in the development of an effective and efficient NGC structure to govern the vehicle behaviour. A deeper analysis on the mechanics and structure of

OpenFISH can be found in [113] and [114]. A photo of the vehicle is shown in figure 39.

FIGURE 39 : OPENFISH AUV, BRAVE VERSION

The purpose of OpenFISH is to design a modular, low-cost and efficient underwater vehicle for general missions of marine exploration. The hardware design reflects these requirements: the system is divided in a series of different interconnected underwater cylindrical hulls, with a common waterproof connector between hulls that shares the low-level communication bus (I2C) and the power signals (that is common for the whole vehicle). If new devices are needed to extend the functionality of OpenFISH and to perform a specific underwater mission, it is just necessary to design just the interior of the hull (some starting models of the mechanics to store the internal component are also present) that must store the new sensors. The bus allows the connection of all the devices to the main controller that implements the NGC system of the vehicle. The main requirements of this system are:

- Connection with different sensors and actuators by means of one communication bus (I2C). The NGC system acts as a Master of the bus;
- Connection with different communication devices (e.g. Wi-Fi, cable and acoustic modem) in a transparent way for the rest of the architecture;
- Control of the thrusters for horizontal and vertical movement, providing auto-depth and auto-heading algorithms;
- Development of a mission model to be downloaded in the vehicle and interpreted;
- Management of internal exceptions and hardware failures;
- Development of the system to be contained in a single NI-MyRIO board.

Since these requirements demand a high degree of modularity and abstraction, a Multi-Agent Architecture was chosen and the robot mechatronic architecture was designed accordingly as shown in figure 40.

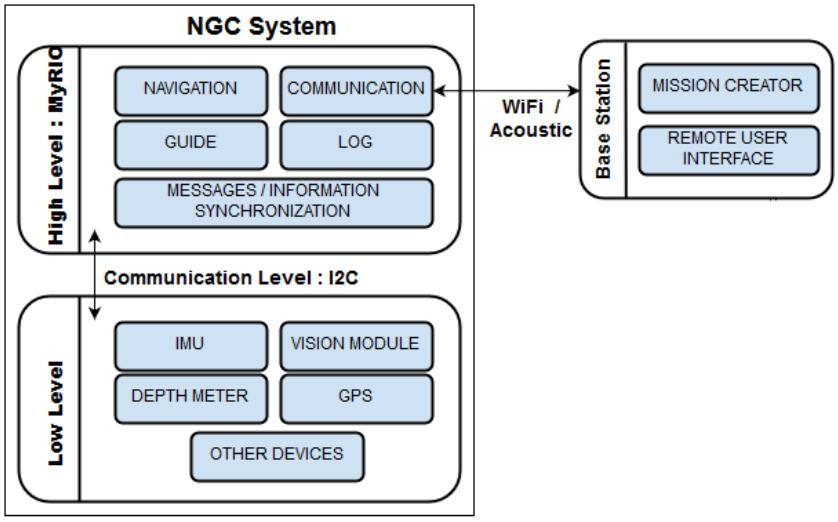

FIGURE 40 : OPENFISH SYSTEM ARCHITECTURE

The architecture of the biomimetic underwater robot can be described as a partially two-tiered vertical architecture composed by two main components: the *Low-Level* and the *High Level*. The union of these two components is the so-called NGC system.

The Low-Level is composed up by an array of devices connected with the rest of the system by means of an I2C bus. There are two types of devices connected: Commercial-Off-The-Shelf (COTS) devices, such as the Inertial Measurement Unit (IMU) the Global Positioning System (GPS) and the depth meter, and custom-made devices, such as the energy, the vision and thruster manager. The custom-made devices have a minimum degree of autonomy in case of emergency by means of tailored hardware and software, which has the duty to take the device into a safe steady-state with a faster and predictable response to a wide range of malfunctions such as low battery or water leaks inside the vehicle. This approach avoids single point failures in the NGC system if there is a major fail related to the High Level.

The High-Level component, implemented on Linux Real-Time device (NI-MyRIO), follows the MAS characterization theory, allowing the increment of the system abstraction and modularity with the design of agents able to fulfil different tasks autonomously. In OpenFISH these techniques are simplified in a point to make the vehicle components purely reactive following the intelligent agent definition

provided in [115]. The developed agents, in fact, lack on proactiveness and social capabilities. Cognitive capabilities are not required either for the type of missions performed by the present vehicle but the control architecture is ready to be connected with other entities such as the ROS compatible robot agency shown in chapter 3.4. Simplification is also needed because of the on-board hardware limitations.

The software architecture is mainly composed by five threads (one for each agent): *Navigation Manager*, *Guide Manager*, *Low-Level Manager*, *Communication Manager* and *Log Manager*. Each agent manages a specific section of the entire vehicle autonomy. The software modules are synchronized with each other to allow for the exchange of messages.

The Navigation Manager agent manages the different functioning modalities: manual guide, automatic guide and emergency. The system starts in manual mode, where the user can manually control and monitor the status of the robot, this mode makes OpenFISH similar to a ROV. If necessary, it is also possible to enable or disable the auto-depth and auto-heading algorithms. The system changes from manual mode to automatic mode when the start command is received from the base station and a mission is correctly loaded into the system. Once the automatic mode is active, the mission executor (that is a set of subroutines of this agent) processes the loaded mission. When the external emergency switch is triggered, or some particular fault event occurs, the entire system enters in emergency mode and all the thrusters are shut down until the external switch is triggered again.

The Guide Manager agent executes the commands sent from the navigation agent related to movement routines such as advance, alignment and reach depth, and transforms them in commands for the thrusters by means of tailored algorithms. When the command is successfully performed or an error is raised, this module sends a feedback to the navigation agent. The guide module also manages the controls loops related to auto-depth and auto-heading algorithms by exploiting data coming from the IMU and the GPS.

The Low-Level Manager agent communicates with all the devices connected through the Low-Level bus (I2C) and keeps the system status information up-to-date. In

addition, the Low-Level module updates the status of all the connected devices that the guide manager has access to. Each device attached to the bus is queried in sequence with a proper scheduler with non-blocking criteria, in order to avoid overlap between requests to different devices. When a new device is connected to the system, the Low-Level module is able to discover it and signals its existence to the rest of the agency. Any agent can read data and interact with the Low-Level devices through a blackboard-type software module that maintains synchronized all the data of the NGC system.

The Communication Manager agent maintains the communication with the base station. Specifically, it transforms the information received and transmitted through two different communication media (Wi-Fi and acoustic modem) in a standardized way, in order to be used simultaneously by the other agents, adding a basic level of abstraction related to the used medium exploited during the mission to provide communication (for example, until OpenFISH is floating it is possible to use the Wi-Fi communication, while it is possible to use the acoustic modem if the vehicle is submerged). The base station has two main purposes: to create, modify and upload a mission (the Mission Creator) and to remotely control the AUV through a custom-made interface (the Remote User Interface). The custom-made interface is used to perform tests, initiate the mission and gather data.

The final agent is the Log manager. It has the duty to write log files in different formats and to save them in a USB pen-drive. The data is recorded along with the timestamp in order to analyse the behaviour of the vehicle and reproduce its movement with other software.

An important aspect is the definition of the *Mission* that the vehicle has to perform. A Mission is defined by different *Tasks* and a mission timeout. When the mission timeout is reached, the mission is over. Each Task has its own timeout and is arranged as a finite state machine. Each *State* of a Task state is composed of a series of elements that allow the system to perform cyclically repeatable actions, as well as actions conditioned by the system status and its global variables, like depth and heading, along with battery and thrusters' operative status. For example, one element of the state structure represents a single vehicle moving action to be sent

to the guide module. Before processing the following element, the mission executor waits for the end signal from the guide module. The mission is editable and loadable to the vehicle through a proper graphical user interface (GUI) and it is exchanged and stored as a JavaScript Object Notation (JSON) string. When all the states of a task have been executed or the task timeout is reached, the mission executor processes the following task. When the mission reaches a timeout or all the tasks have been handled, the automatic mode manages the surfacing action of the vehicle. Once the vehicle is out of water, the system switches to manual mode. If a critical condition is experienced, such as low battery or a hardware failure, the mission terminates.

A version of OpenFISH (called BRAVe: Biomimetic Research Autonomous Vehicle [114]) was developed to participate to the Student Autonomous Underwater Competition – Europe 2016 (SAUC-E) powered by NATO where the vehicle, thanks to its unique features related to the propulsion system and its modularity won the Innovation Award.

The OpenFISH project, that will be a good scenario to implement the designed robotic infrastructure, is still in progression and some of its features will be enhanced with the new solution that will be further designed and developed:

- The functional structure in five agents is good and viable for the new system in terms of division of responsibilities (communication, low-level interface, navigation, guide and logging). Unfortunately the hardware and software limitations made the system not suitable to be implemented in other similar scenarios;
- In this development, only I2C has been used. To increase the modularity of the system, more low-level buses and interfaces should be used. Some examples could be a serial link (that usually is exploited in the umbilical cable to communicate with underwater vehicles or could be used to interface with the Payload Units of the DocuScooter), CAN, and ROS (to easily connect third-party software and hardware).

3.3 The Home Automation System (HAS)

One scenario of application of the MAS theory in the studies conducted in the LabMACS laboratory was the domotic environment. The output of this study was the development of a Home Automation System (HAS) [64] to manage the connection of a number of appliances and devices for house management, which are connected by a communication line of some kind.

Among the tasks of the home automation system, those of major concern at the present stage of development consist in:

- regulating energy consumption, in order to avoid peaks in the electric load that may exceed either fixed or time depending thresholds and, possibly, planning the use of energy according to tasks and to time varying cost;
- monitoring the behaviour of different appliances and, possibly, detecting and signalling malfunctions or failures;
- facilitating the interaction with human users by allowing remote control, planning and monitoring.

From the underlying MAS theory [44], an agent is defined as a virtual or physical entity able to possess, up to different degrees, the following capacities:

1. it can perform specific actions in a given environment;
2. it can perceive elements of the environment;
3. it can construct (partial) models of the environment;
4. it can use personal resources and environmental resources;
5. it can orient its actions toward specific goals;
6. it can communicate directly with other agents in the environment;
7. it can offer services to other agents;
8. it can govern its action per its possibilities and limitations, to its goals, to its knowledge of the environment, to the resources available in the environment.

From this list of capacities two specific types of agents: the Domotic Object (DO) and the Domotic Agent (DA). The first that has at least the general capacities 1, 4, 5 and 8 and, concerning capacity 6, it can communicate to other agents in the

environments at least its requirements about environmental resources. The latter is a DO that has at least in addition the general capacities 2. A domotic agent is called *Cognitive* if it has also capacity 3.

General control strategies for the above described systems consist of a set of *Rules* that establish priorities in gaining access to limited resources on the basis of the available information. Such rules are assumed to have been synthesized in such a way to maximize a functional that, more or less abstractly, describes user's satisfaction.

A representation of the proposed scenario is shown in figure 41:

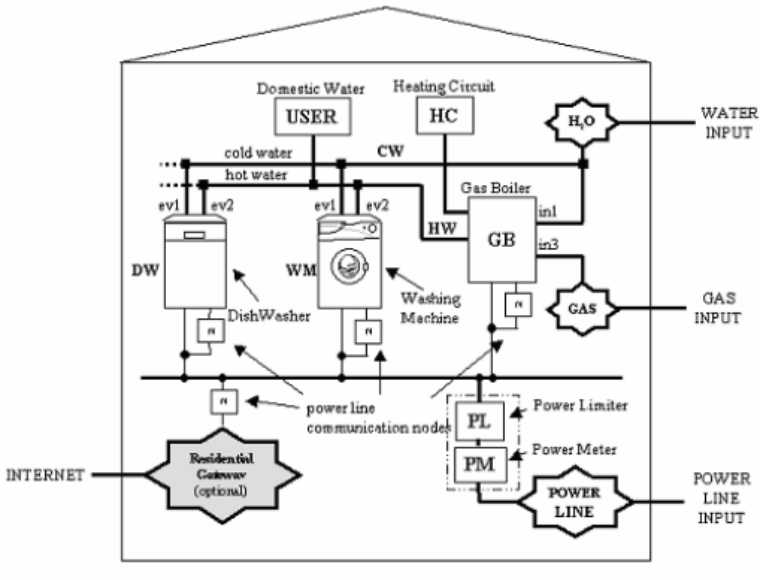

FIGURE 41 : THE HAS SCENARIO FROM [64]

We can recognize four different aspects:

- **Agents**: in this scenario agent are represented by a human and by the different domotic devices (washing machine, dishwasher, gas boiler, heating system, power meter and power delimiter). Other different devices could be added if needed. The presented devices are the ones that are employed during the simulations and tests;
- **Resources**: the domotic environment has three basic external resources: *cold water*, *gas* and *electricity*. Hot water, which is viewed as an additional resource, can be produced internally, by transforming other resources. The available amounts of cold water and gas at a given time are assumed to be

free from limitations, while the available amount of hot water and that of electricity at a given time are subject to limitations. The Gas Boiler (GB) supplies hot water both for sanitary use and for heating purposes through the Hot Water Circuit (HW). The Heating Circuit (HC) and the Human User (USER) can obtain hot water from the Hot Water Circuit. The washing machine and the dishwasher can both use hot water produced by the boiler, competing between them, with the HC and with the User(s) in the exploitation of this limited resource, or they can produce hot water for internal use employing electricity and cold water;

- **Communication lines:** Information flows on a communication line that exploits power line and it is connected to a residential gateway. With this latter component, data can be exchanged for remote assistance or control. In normal operation, each agent uses a part of the available resources, and clearly, since the availability of two of them are limited, it is useful to have an allocation strategy that can be implemented by the various agents. So, the agents must be able to coordinate their functioning by means of exchange of information.
- **Nodes**: the communication through the power line is made by exploiting suitable interconnection devices called Nodes (N). Their basic function is to measure the actual energy load imposed by the appliance they are connected to and to make this data available through the communication network they are linked to. In addition, the nodes could be endowed by the capability to read some data from the appliance they are connected to and transmit to the appliance commands or data coming from other nodes of the network, per the ability of the appliance to establish a dialogue with the node. The nodes are also capable to process the data they collect by means of an internal microprocessor and a set of software instructions. In this structure, the Nodes incorporate part of the intelligence of the systems and, individually or together with the appliance they are connected to, they can implement the control strategies which optimize the global system performances.

Referring to the system we have described above, the problems in regulating the overall rational behaviour are:

- Distribution of limited resources, like electricity and hot water according to specific priorities in order to avoid unpleasant events for the user (for example, in case the user is using hot water for taking a shower, this action should have a higher priority compared to the washing machine that need how water in order to work);
- Organization and scheduling of the operation of different appliances, in order to keep the global electric load with the supplier's established limits, in order to avoid an unexpected shutdown;
- Planning related to the use of electricity and gas that must take in account economic priorities;
- Optimization of the performance of individual appliances in relation to specific criteria of user's satisfaction, and the previous constraint imposed by strict requirements.

After having defined the principal elements that form the overall system, we can give the following definition of HAS. A HAS consists of the following elements:

- A set GR of Global Resources;
- A set DO of Domotic Objects;
- A set DA of Domotic Agents, subset of DO;
- One Information Network IN, that connects domotic objects and agents;
- A set R of Rules that govern the individual behaviour and the concurrent operation of domotic objects and concerns:
  - use and transformation of external resources,
  - communication,
  - perception and understanding;
- A set L of operators, called Laws, which describe the time evolution of the global system according to the individual behaviour of objects and of agents.

This characterization of the notion of HAS agrees with the general point of view of MAS theory. The behaviour over time of the HAS agents are described on the basis of previous definition is completely determined by L and it depends, in particular, on the Rules which form R. Then, it is possible to study the effects of different choice of the Rules that form R on the global evolution and behaviour of the system and, in

particular, to evaluate its performances in terms of functionals that represents user satisfaction.

This formalization can help to design and develop framework that, in addition, being able to track and foresee the global behaviour, it is able to detect and analyse critical parameters of the systems by means of simulations procedures.

The study related to the HAS was a very preliminary one, but some of its main concepts are still valid as requirement for the that will be designed and developed:

- **Environment and resources**: The concept of environment is not directly modelled, but the idea of *Global Resources* will be an important component of the environment, that will act as a shared space where agents will be able to coordinate considering limited resources (if any);
- **Different types of agents**: Two types of different agent are introduced, that take part in the same agency: the *Domotic Object* (that acts as a reflex agent with a reactive architecture) and the *Domotic Agent* (that acts like a utility based agent with a BDI/Hybrid architecture). In the infrastructure that will be developed, it will be important to define different agent schemas, endowed with different capabilities, but all of them will be able to contribute at the same agency;
- **Simulation environment**: when there is the analysis of a complex system, it is fundamental to have tailored tools that ease the debug process. Moreover, if the system is formally defined, it is possible to have a full simulation environment that could be used to detect anomalies in a safe environment.

3.4 MAS for general purpose ASV

In the recent years, with the objective to develop a low-cost and effective solution to perform support to marine researchers, LabMACS designed and developed an ASV (Autonomous Surface Vehicle) composed mostly by COTS components for general purpose missions. This solution is able to avoid the elevated cost for a supply vessel and its crew, allowing a modular approach to different similar problems, easing the work related to add new hardware and software architectures in the system. The developed platform can be endowed with tasks to perform different inspection and

research missions. An extended explanation of the whole architecture functioning can be found in [116]. One possible scenario is shown in figure 42.

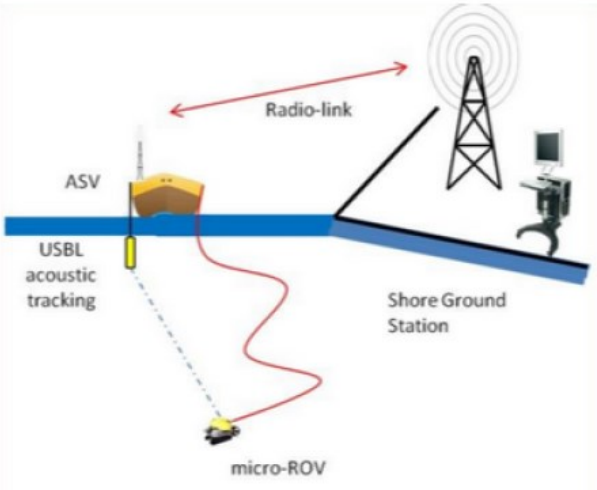

FIGURE 42 : AN EXAMPLE SCENARIO FROM [116]

In the proposed scenario, the user from a Shore Ground Station, using a wireless link, can remotely deploy and teleoperate a micro-ROV in a point of interest, while the Autonomous Surface Vehicle (ASV) is endowed with the task to autonomously navigate to the defined point in the ocean, to manage the deployment of the micro-ROV, to maintain formation with the underwater vehicle by means of a USBL tracking device in order to keep a safety distance from it and to recover it at the end of the mission.

This scenario could be easily modelled as a MAS problem, where the different components (the micro ROV, the ASV and its constituent components) could be intended as autonomous agents with different tasks to perform towards the common goal to perform the mission.

Focusing in the ASV, its mechanical, hardware and software components have been constructed using commercially available, low cost Components-Off-The-Shelf (COTS). The considered ASV is a mono-hull boat with a unique steering, outboard electric motor, differently from the most part of existing literature, that is based on a catamaran-like double hull ASV.

The mechatronic structure of the ASV is composed by a set of *subsystems*, consisting of hardware and software components. The various subsystems can be described as follows:

- **Central Control Subsystem**: Host an IMU and a GPS device, which are used to evaluate position, orientation, velocity and acceleration of the ASV, and a video-camera, with motorized pan and tilt mount, are directly connected a Single Board Computer (SBC).
- **Engine subsystem**: It is composed of the outboard electric motor and by a custom board that governs it.
- **Rudder subsystem:** It is composed of a mechanical steering system, which is actuated by a stepper motor endowed with an incremental shaft encoder. The stepper motor is closed loop controlled by a microcontroller.

Each subsystem, from an operational point of view, is divided in three levels, called, respectively, the *Agent Level*, the *Interface Level* and the *Hardware Level*.

The Agent Level refers to software components, called Agents, which are organized per a Multi-Agent System (MAS) architecture that exploits the Robot Operating System ROS [117] as communication middleware. In each subsystem, the software agents take care of the high-level tasks and of the communication with other entities in the ROS framework. This solution allows the subsystems to interact between them by exchanging data by exploiting ROS and to perform specific tasks in response to external inputs in a coordinated way. The Interface Level refers to the low-level software routines that interface each agent with various I/O devices. The Hardware Level refers to actuators, sensors, computing devices and other electronics components.

The agents and the low-level software routines are implemented on computing devices of various kind that form the ASV computing structure. Each computing structure is composed by two layers: the *High-Level Layer* and the *Low-Level Layer*. The first is composed by an ARM board with a Linux distribution. This level has the duty to run the agent and the related ROS infrastructure. The second is composed by a microcontroller, which has tailored routines to manage the underlying hardware.

In practice, the ARM boards are used to execute behavioural and decisional tasks at the Agent Level and at the Interface Level, while the micro-controller boards implement the control strategies that govern actuators and sensors at the Hardware Level. The advantages of this architecture are the high computational capabilities to run decisional and ROS protocols provided by the ARM boards and the reliability of control performances assured by the use of dedicated microcontrollers. The two boards communicate between them using a serial communication link and an error handling protocol.

In this implementation, agents are self-sustained routines that are able to control their life cycle, which is shown in figure 43. The most important phase of the execution is the "processing" phase, where the agent uses different *Behaviours* to operate in the agency. The agents, together, form the so-called Boat agency.

FIGURE 43 : AGENT LIFE CYCLE

These are the agents implemented in the ASV:

- **Master agent:** this agent coordinates the MAS in the ROS framework; it is used to monitor the status of the infrastructure and to alert about MAS failures (failure of the ROS middleware or shutdown of the Boat agency) and explicit shutdowns;
- **Engine agent:** this agent implements the open-loop control of the angular speed of the thruster of the outboard electric motor;
- **Rudder agent:** this agent implements the control of the steering angle of the outboard electric motor;
- **GPS agent:** this agent acquires and publishes, the position of the vehicle obtained by the GPS;
- **IMU agent:** this agent acquires and publishes attitude, rotational speeds and accelerations of the vehicle obtained by the IMU;
- **Controller agent:** this agent implements the Navigation Guidance and Control (NGC) procedures of the ASV such as autonomous and remote

navigation. This agent exploits the Engine, Rudder, GPS and IMU agent to operate the whole ASV;

- **Camera agent:** this agent streams video from an on-board surveillance IP-Camera and manages pan, tilt and focus according to external requests. This aids the teleoperation of the ASV;
- **Logger agent:** this agent logs the data published by the agents in a file stored in the disk;
- **ROV agent:** this agent implements the control and sensor of the micro-ROV;
- **USBL agent:** this agent implements the driver for the USBL tracking system by publishing the position of the underwater micro-ROV.

The functioning of an agent is regulated by its *Behaviours*. A behaviour allows to an agent to interact with the environment or between them within the agency. The life cycle of a behaviour, shown in figure 44 is managed by the agent itself.

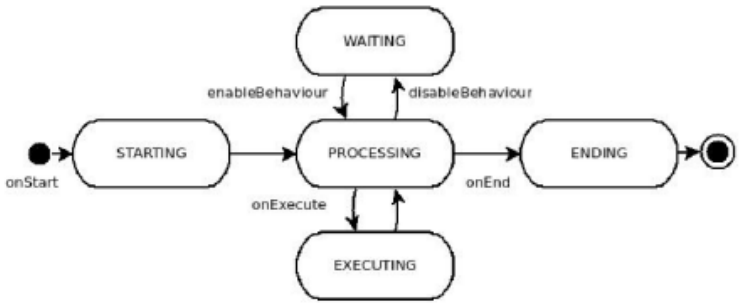

FIGURE 44 : BEHAVIOUR LIFE CYCLE

Agents can exhibit four types of behaviours:

- **OneShotBehaviour**: one or more actions associated to the behaviour are executed only one time and instantly;
- **CountdownBehaviour**: one or more actions associated to the behaviour are executed only one time after a given delay;
- **CyclicBehaviour**: one or more actions associated to the behaviour are executed periodically with a given period;
- **EventDrivenBehaviour**: one or more actions associated to the behaviour are executed only if and when a specific event is triggered. The possible events are: notification of a new message, notification of a service request

and cancellation of subscription by an agent on which the one at issue is dependent.

Dependencies are another important feature of the system, and it is a way to coordinate agents' behaviour when a related agent stop its normal functioning. When a dependency is defined between two agents, if the first leaves the agency or its functioning is not corrected, the dependent one triggers a EventDrivenBehaviour to handle this eventuality. This feature is usually exploited to keep a safe state in the dependent agent.

The communication between agents is performed through ROS topics, which acts as a middleware between agents. ROS uses the publish/subscribe messaging pattern where senders of messages, called publishers, do not send directly data to specific receivers, called subscribers, but instead use the concept of topic to categorize topics without any knowledge if there is a subscriber that receives the message. Each agent has the duty to make their existence clear in a well-defined topic, in order to coordinate with other agents. In addition, the exchanged data is also published in ROS topics. There is no specific mechanism to deliver a message from an agent to another one, the data is freely available in the ROS framework. The content of these messages is managed through a ROS message definition.

In [116] and [118] some experimental results are provided: the ASV was tested to perform different tasks in different modalities (remote control, different types of path following, target tracking), while simulating failures between agents, in order to affirm its robustness in different situations.

The architecture and technology used for the ASV has proven a valid starting point to be enhanced, from different points of views:

- **Organization architecture**: the developed agency acts as an Aware ACMAS architecture. There is no explicit definition of an organizational model, but other agents are aware of the presence of the other relevant agents by means of the exploiting of the Dependency concept, acting in some cases as a mechanism of indirect coordination(stigmetry). The idea is to implement an Aware OCMAS architecture;

- **Agent architecture**: it is difficult to implement a real Goal oriented agent with the architecture of agent proposed. It could be possible, however, to have this typology of agent in the same Agency together with agent with other architectures;
- **Agent life-cycle management and control**: there is no centralized way to manage the agent life cycle: the idea is that, for each device able to run agents, there must be a component that is able to manage, in a platform-wise manner, their life-cycle in order to obtain a more robust and modular system;
- **Dependency links**: the management of the Dependency between agents is defined inside the agent, and is in the sake of the programmer to manage it correctly. A more explicit way to exploit the Dependency between agents in an organization is by means of the concept of Roles and Links between them. If in a group there are some specific roles one of them is missing and there is some explicit dependency between two of them in the organizational model, the dependent one is aware of the missing role;
- **Abstraction of connected hardware**: there is a strong separation between the Agent Level and the Hardware Level by means of the Interface Level, but there is no standardization of the last two levels, like the one designed for the DocuScooter by means of Payload Specifications. Moreover, some of the agents (e.g. IMU and GPS) could be modelled as RobotSensingParts and not as real agents. By the other hand, modelling these components as agents, there is the advantage to ease the access to the data provided to other agents through ROS topics;
- **Abstraction of communication interfaces**: The employment of ROS has advantages and disadvantage. By a way, it is possible to interface the developed architecture with a high number of third-party hardware and software, but at the actual state, ROS is the only communication channel available. The idea is to provide a common communication interface as abstraction of more, different, concrete Low-Level communication interfaces (RobotCommunicatingParts). This will allow to achieve systems

where Robots can communicate through different channels in a transparent way.

3.5 LabMACS integrated system architecture

In the recent years, in the LabMACS laboratory we started to design a modular vertical-layered architecture to allow interconnection between different surface and underwater technologies from different vendors in order to obtain provide a wide typology of available setup for different marine missions such as 3D reconstruction for archeologic purpose ([119], [120]) and biological data acquisition and gathering. The process of standardization of technologies arranges them by the role that they can acquire in the system. So, for example, different vehicles or technologies can acquire externally the same role, performing analogue actions that provide the same kind of information. This process allows to design different formal processing pipelines that transfer, gathers and transforms the input data in a requested output.

The design is inspired by two main theories:

- **System of Systems (SoS)** [121]: in this approach, a global system is composed by the large-scale integration of different self-contained systems with the goal to satisfy global needs in a more efficient way.
- **Internet of Underwater Things (IoUT)**: in this approach, that is a specialization of the IoT theory, numerous devices are connected in a wide distributed network that can collect robust and localized data from the marine environment in an autonomous, modular and extensible way in which the modules are able to self-organize accordingly.

A schema of the proposed architecture is shown in figure 45.

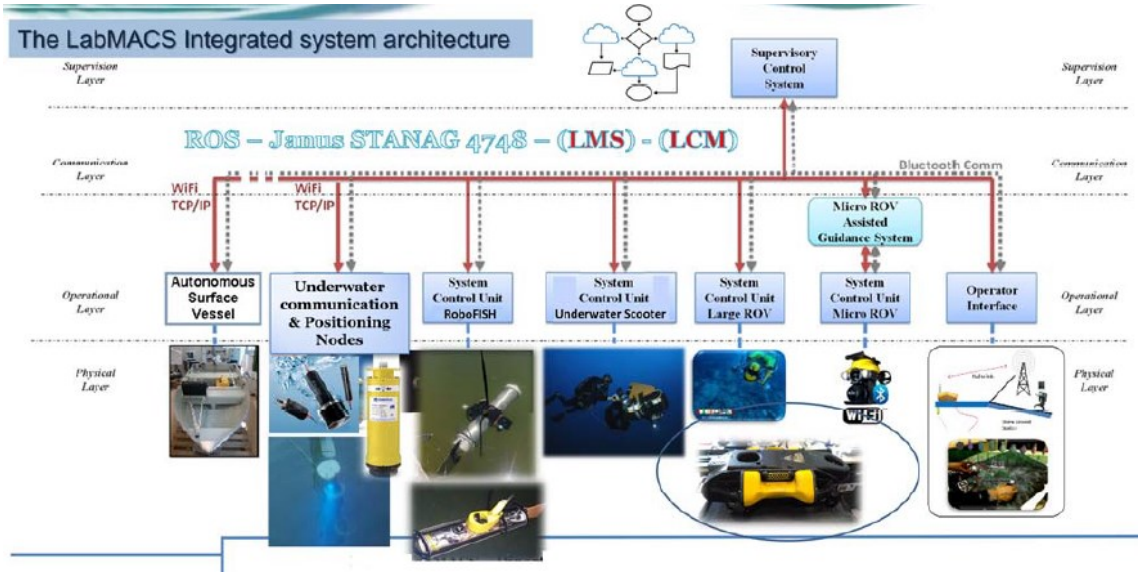

FIGURE 45 : LABMACS INTEGRATED SYSTEM ARCHITECTURE

In the previous image. it is possible to notice four different layers:

- **Physical Layer:** this level includes all the available the commercial or custom hardware that must be able to cooperate and coordinate through the system in a transparent way, my making a unique agency of agents. In the previous image it is possible to notice some of the integrated technologies, such as:
    - A custom made ASV. It is the same ASV of the previous case study, but it is possible to use it with the rest of the infrastructure;
    - Ultra-Short Base Line(USBL) devices that allow underwater positioning of mobile underwater targets;
    - Custom made underwater nodes that provide communication through optical transmission;
    - The DocuScooter used as an underwater data acquiring device supporting the activities of scuba divers;
    - different commercial ROVs of different sizes: small micro ROVs (VideoRay Pro4) and a large working class ROV;
    - A shore ground station where the end users can use different technologies such as command consoles, mission planning and monitoring application and control devices like a smart force feedback joystick [122] that helps amateur pilots to better control an underwater vehicle;

- **Operational Layer:** this level is endowed with the task to separate the hardware implementation from the higher layers. It embodies the routines to control and manage the underlining Physical Layer while providing its access from outside. This layer is implemented by means of commercial or custom hardware solutions (servers, PCs, Single-Board computers or microcontrollers) that exploit different kind of communication media to exchange information.
- **Communication Layer:** this layer allows the transmission of information defined in a formal way by means of different communication channels and protocols, allowing to add and remove devices in a modular way. Some examples are:
    - **Wireless networks:** Wi-Fi (by means of TCP/UPD sockets), Bluetooth, LoRa;
    - **Wired networks:** Ethernet (by means of TCP/UPD sockets), Serial channels (RS232, RS485 and UART), CAN bus and I2C;

  Some used protocols are the LabMACS protocol, ROS [117] and Janus STANAG while others (LMS and LCM) are currently in development. The first protocol, in particular, is designed for efficient serial communication, and it is similar to the one used for the DocuScooter platform;

- **Supervision Layer:** this layer acts as a headquarter of the whole infrastructure where each connected device at the Operational Level is visible and its status is constantly monitored. This level allows to:
    - Monitor the status and operation of the whole architecture by means of a list of performance indexes. Indexes allow to statistically analyse the situation while providing easily readable parameters to evaluate the overall operating condition and mission;
    - Assign global tasks and goals.

The SoS concept is similar to the MAS theory, and this analogy has been already analysed [123], because they have, in overall, the same objective: generate an infrastructure able to solve problems in a collaborative, coordinate and efficient way. The main differences are related on:

- **Nature of the entities**: in the SoS theory, the elements are concrete technologies, initially not designed to be interconnected. On the other way, in the MAS theory, the entities forming the infrastructure are essentially software agents, in which the social nature is embedded in the concept of agent itself.
- **Main effort**: in the SoS theory, the effort is directed to seamlessly integrate different technologies in order to allow interoperation and control between different components of the infrastructure, starting from a lower level of integration. In fact, different systems that use different protocols and technologies to communicate need to be translated in a subset of protocols able to interoperate in the same network. On the other way, in the MAS theory, the effort is directed to allow integration to a functional level, by allowing seamlessly communication between different entities, thanks to the fact that the scenario is higher levered, with abstractions from the lower and concrete level.

The idea is that the infrastructure that will be developed should be able to merge transparently with the proposed infrastructure, providing different roles that can be inserted in different processing pipelines, in relation to the overall mission of scenario that the system is situated.

Chapter 4: Design and characterization of models for Robots and Multi-Robot Systems

In order to express in a more formal manner some aspects of the Robots and of a whole organization of robots, a great effort has been put in the definition of different models able to define different aspects of this type of problem. The usage of models has many advantages such as:

- Concise representation;
- Usually provides a high level of abstraction;
- Possibility to reason on these models to take proper actions;
- Usually defined in a readable format and platform independent, allowing to be easily shared and interpreted by different devices;

Three models are presented in this dissertation, each of them representing a different aspect of the Robot: the *RobotPart Model*, the *Skill Tree Model* and the *Organization Model*.

The *RobotPart Model* is used to provide a first abstraction of the hardware connected in the robots by defining different classes of components (called RobotParts) that can be connected to the robot. If two different components apply to the same RobotPart class, external software can use this definition to efficiently gain access to the component, regardless to the real nature of the hardware connected.

The *Skill Tree Model* is used to define which are the roles that a specific robot can play in an environment. The applicable roles are defined by which components are currently attached. The list of requirements to play a role can, for example, change between different instances of the infrastructure.

Finally, the *Organizational Model* is shared among all the infrastructures in the network and it is used to manage different three different aspect of the organization:

the Environment where the robots are situated, the Mission to be completed and the list of Roles that can be played in the organization.

4.1 RobotPart Model

A Connector of the infrastructure can register one or more RobotPart to the Platform. The concept or RobotPart follows the idea explained in [2], where a RobotPart is defined as a Device used by the Robot to interact with the environment. Four classes of RobotPart are included:

- **RobotSensingPart:** defines a type of sensing device that is able to sense the surrounding environment. An example could be a IMU or a GPS receiver. This part can provide different sources of information called Topics.
- **RobotActuatingPart:** defines a type of actuating device that is able to act with the surrounding environment. An example could be a thruster or a light that can be toggled. This part, in a way similar to the previous RobotPart, can provide different sources of information called Topics.
- **RobotCommunicatingPart:** defines a type of communicating device that is able to send and receive messages through a network.
- **RobotProcessingPart:** defines a type of processing device that is able to process information. It can act as a bridge to some external or internal data processing. This part provides different processes called Services that can be called. An example could be some tailored function provided by a third-party software.

The model of a RobotPart is implemented as a tuple with these fields:

<Name, Description, Type, [Decorator]>

Where *Type* can be *SENSING*, *ACTUATING*, *COMMUNICATING* and *PROCESSING*. In the first two types, also the field *Topic* is included, while in the PROCESSING case, the field *Service* is included. *[Decorator]* is a list of *Decorators*. *Topic* is a field composed by an array of tuples defined as:

<Name, [Params]>

Where *Param* is defined by a couple:

<Name, Type>

Where Name is a unique string, while Type is a value included in a group formed by *INTEGER*, *FLOAT*, *BOOLEAN*, *STRING*, *LIST* or *OBJECT* or is a name of an instance of *Custom Type* (explained below). When the Type is LIST or OBJECT another element of the tuple is allowed, represented by a list of Params.

Service is a field composed by an array of tuples defined as:

<Name, Request, Response>

Where Request and Response have the same format of Topic.

A Decorator is an additional and optional property of the RobotPart. Its semantic is not defined, its interpretation is a duty of the higher levels of the framework.

The RobotPart of a Robot, together, create the *RobotInterface* of the Robot. There is no limitation to the number of RobotParts that a Connector can provide. For example, a connector able to manage a CAN bus could register different actuating, sensing and processing parts.

4.1.1 Custom Type Model

Custom Types help to reduce the amount of code that is used to define RobotParts. They define a set of commonly used types of messages. Some rules are used to avoid complexity:

- Each Custom Type must have a unique name;
- Only the fields *INTEGER*, *FLOAT*, *BOOLEAN*, *STRING*, *LIST* or *OBJECT* can be used as *Type*;
- It is not possible to define a Custom Type through another custom type (this is a consequence of the previous rule).

These are some examples of some used Custom Types:

- **2DPos:** an object composed by two FLOAT fields: X and Y;
- **3DPos:** an object composed by three FLOAT fields: X, Y and Z;
- **3DOri**: **a**n object composed by three FLOAT fields: Roll, Pitch and Yaw;
- **GPSPos:** an object composed by the couple Latitude and Longitude.

4.2 Skill Tree Model

Different robot can play different roles, and this depends from two conditions: it has all the required RobotParts to achieve this role and it knows "how" to do it. For example, if a "Grabber" Role is available in the organization, only a robot that has a manipulator can play it and should have tailored software and procedures to manage this role. While the second condition relies to the ability of the Agent Layer to assure the maintenance and correct usage of the manipulator, the first condition is ruled by means of a Skill Tree (ST). Essentially, a ST is used to transform a set of decorated local RobotPart to available roles that the can be played.

This representation is elaborated by the Organizational Module together with a list of vacant Roles, and the list of actual RobotPart. The ST is used to decide if the infrastructure has all the requirements to create an agent to fulfil a vacant Role. Of course, there must be coherence with the Role defined in the Organizational Model and the one defined in the Skill Tree.

The model, shown in figure 46 (while, in figure 47 an example for the "grabber" role is shown) shows three different elements: decorated RobotPart, Skill and decorated Role.

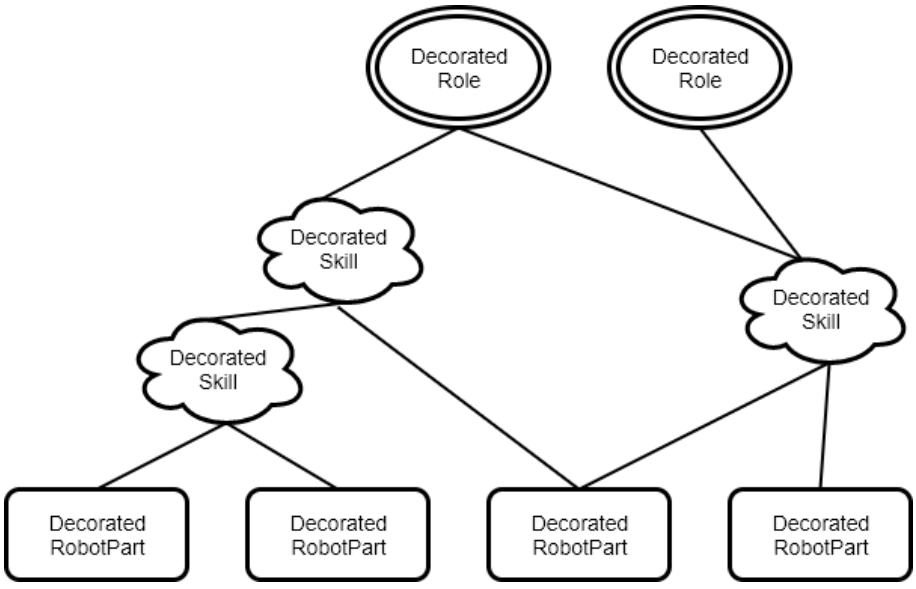

FIGURE 46 : SKILL TREE STRUCTURE

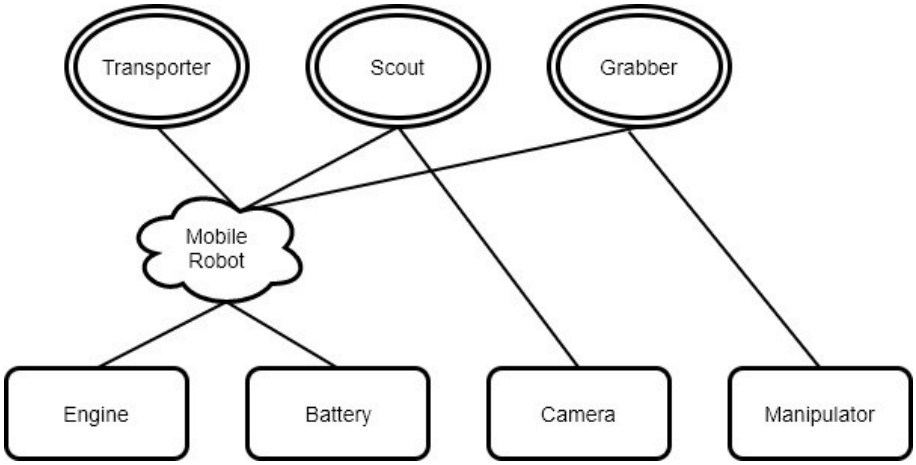

FIGURE 47 : SKILL TREE EXAMPLE WITH "GRABBER" ROLE

- **Decorated RobotPart:** this element represents a single RobotPart of the local RobotPartInterface with the addition of decoration properties added to the element. These properties are used only at this layer and they have no implication to the lower layer and they can modify how roles are assigned. These are the allowed decorations:
    - **Unique:** if this keyword is used, only one role can use possess the RobotPart. This allows, for example, to lock the "Engine" RobotPart of the platform to just one role with the keyword "unique" to the RobotPart;
    - **Optional:** if this keyword is used, the connected decorated skill or role can be achieved also without this specific RobotPart.
- **Skill:** this element acts as a grouping concept, defining skills that the robot achieves by possessing some sub-skills or robot parts.
- **Decorated Role:** this element represents a single applicable role with the addition of decoration properties. A Decorated Role can be composed by different skills and Decorated RobotParts These are the allowed decorations:
    - **Group:** this property specifies that a role can be achieved if and only if any RobotPart that composes it has the property "Group" that has the same value. This is helpful for example in a simulation environment, where a single Connector could be able to define different instances of the same type of RobotPart;

- **Priority:** this property modifies how the algorithm that analyse possible roles tries to satisfy them. A Role with this property will be tested before others, so there is a higher chance that it is fulfilled.

4.3 Organization Model

The main concepts of the Organization Model are arranged as shown in figure 48. From the top of the structure three subparts that regroup different aspects of the organization are defined: Environment, Role and Mission.

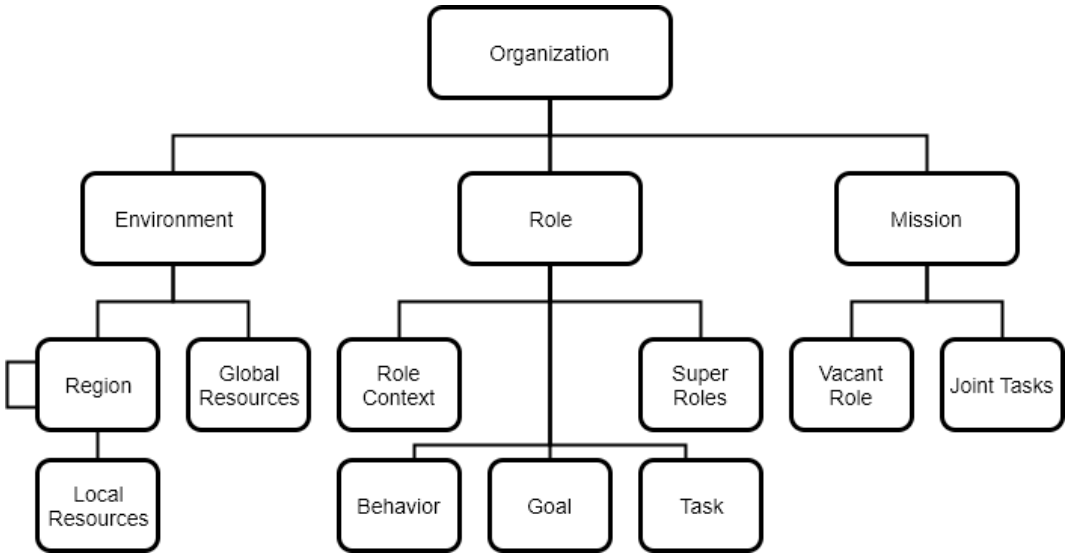

FIGURE 48 : THE ORGANIZATION MODEL

Environment

This subpart of the model defines the static properties of the environment where the agents act. The environment is essentially defined by the following couple:

<[Reg], [GR]>

Where [Reg] is a set of *Regions* and [GR] is a set of *Global Resources*.

A *Region* is a logical and/or spatial subdivision of the environment where the robots are situated and operating. At least a region must be defined. For example, in a complex scenario composed by different water and land vehicles that must cooperate, it is possible to define at least two regions: “water” and “land”. A region is defined by the tuple:

<Name, [Reg], [LR]>

Where *Name* is a unique string defining the region, [Reg] is a ret of sub-Regions and [LR] a set of *Local Resources*. Cycles among regions are not accepted. As example, two sub-Regions of the previous "water" region could be "submerged" and "surface".

A Global Resource is a Resource that is shared among all regions while a Local Resource is shared in the related Region and in its sub-Regions.

A Resource is defined by the following tuple:

<Name, Type>

Where *Name* is a string defining the name of the resource and Type a value between *INTEGER*, *FLOAT*, *BOOLEAN*, *STRING*, identifying the type of data stored.

Role

This subpart of the model defines the different roles that an agent can play in the infrastructure. An agent can play one role at the same time, but more agents can run in the same platform.

A Role is defined by the following couple:

<RC, [B], [G], [T], [SR] >

Where RC is the *Role Context*, [B] a list of *Behaviour*, [G] a list of *Goals*, [T] a list of *Tasks* and [SR] is a list of *Super Roles*.

The *Role Context* is a list of *Properties* that the agent must share to the rest of the agency and they define a current external state of the role that the agent is playing. For example, a Robot that has the Carrier role that is endowed with the role to transport some objects, could use a Property in the Role Context to share the actual count of elements that it is carrying.

A *Property* is defined by the following couple:

<Name, Type>

Where *Name* is a string defining the name of the property and Type a value between *INTEGER*, *FLOAT*, *BOOLEAN*, *STRING.*

A *Behaviour* is a functionality that can be invoked for a specific role. The Behaviour can be called by another agent in an autonomous way or as inside a joint mission between more agents. It is expressed by the tuple:

<F, [ReqA], [ResA]>

Where F is the *Functor* of the command to invoke, representing the name of the function to call, [ReqA] a list of *Request Arguments* used as input if the command to invoke and [ResA] a list of *Response Arguments* used as outputs of the command to invoke.

A *Goal* represent a state that the role should achieve. How to reach it is dependent on the agent that plays that role. It is expressed by the tuple:

<N, [Args]>

Where N is the *Name* of the Goal, while [Args] a list of *Arguments* used as parameters to configure the goal. For example, a Goal *GoTo* could have as parameters the position to reach in order to fulfil the goal. During runtime, when a role is endowed with the task to fulfil a Goal, it can have four possible states:

- *INACTIVE:* this is the default state, and signals that the agent is not trying to fulfil this goal;
- *REACHED:* this state is obtained when the goal is successfully reached and satisfied;
- *PROCESSING:* this state is obtained when action is currently performed by an agent that is playing the role to reach the Goal;
- *FAILED*: This state is reached when it is impossible to satisfy the goal. Usually this last state requires a replanning of the future actions.

A *Task* represents a subroutine of the role, composed by well-defined set of behaviour of the role to perform.

Finally, a *Super Role* is the identifier of another role and it is useful to combine Behaviours and Role Contexts in a modular way. For example, if a generic Role "Robot" with a unique behaviour "move" that allows movement, is a Super Role of another Role, this latter can use the "move" behaviour. It is important to notice

that, by convention, corresponding behaviours, goals, tasks and properties of the Role Context are not overridden with the ones of the Super Roles.

Mission

This subpart of the model defines all the information related to the mission that the different agents, that play a role, should fulfil. It is defined by two elements: a list of vacant roles and a Joint Task.

Vacant Roles are Roles required to the fulfilment of the mission, and agent must apply to a vacant role in order to perform the mission playing in the agency. It is defined by the tuple:

<Name, Role, MinCard, MaxCard>

Where *Name* is a unique string defining the Vacant Role name, *Role* is the identifier of the corresponding role, *MinCard* is the minimum required cardinality while *MaxCard* is the maximum allowed cardinality. If the minimum cardinality is 0, the Vacant Role is defined as *optional*.

A *Joint Task* is similar to a task of a single role but the elements combined in the behaviour tree concerns all the available roles.

Chapter 5: Design and characterization of the middleware of a Multi-Robot System

With the experience and feedback received from the previous technologies and the study of the actual state of the art of Robots, Agents, Multi-Agent Systems and Environment modelling, in this dissertation a general-purpose infrastructure for exploration of non-structured environment and coordination of different robots in a shared environment is designed and presented.

The general requirements of the infrastructure are:

- **General purpose**: the solution proposed must be viable for different simulated and concrete vehicles, tailored for different environments (sea, surface and land) that must exploit the same infrastructure to communicate and coordinate;
- **High modularity**: the solution proposed must allow the connection and recognition of different components attached to it, without compromising the entire system and by modifying its behaviour if necessary. Because of this, a unified way to connect third-party and custom external hardware components must be defined;
- **High extensibility**: the solution proposed must allow to add new compatible components or functionalities easily. This is reached by means of the definition of interfaces between software modules and the formulization of models to represent the different components;

First of all, the overall infrastructure will be presented, then each layer will be deepened in order to present the fundamental components and objectives for each one.

5.1 The infrastructure layout

The overall infrastructure is shown in figure 49. It is a four-tiered vertical architecture divided in the following layers:

- **Connector Layer (CL):** this layer groups the different *Connectors* that are attached to the Robot that provide access to the hardware connected to each Connector.
- **Platform Layer (PL)**: this layer defines the component that implement the logic that provides abstraction and access to the *RobotParts* connected to the robot and that provides communication between different robot in the network through a unique interface;
- **Organization Layer (OL)**: this layer defines the components that implements the organizational and environment models and logic of the infrastructure. It is the component that decides, together with other OLs, the roles that the robot must play in the environment.
- **Agent Layer (AL)**: this layer is composed by different *Agent Containers* that exploit the interface provided by the Organization Layer to manage the life cycle of different agents and to access to information useful to their operations. There could be numerous Agent Layer instances in the infrastructure, each of them providing a different set of agents or ways to implement them.

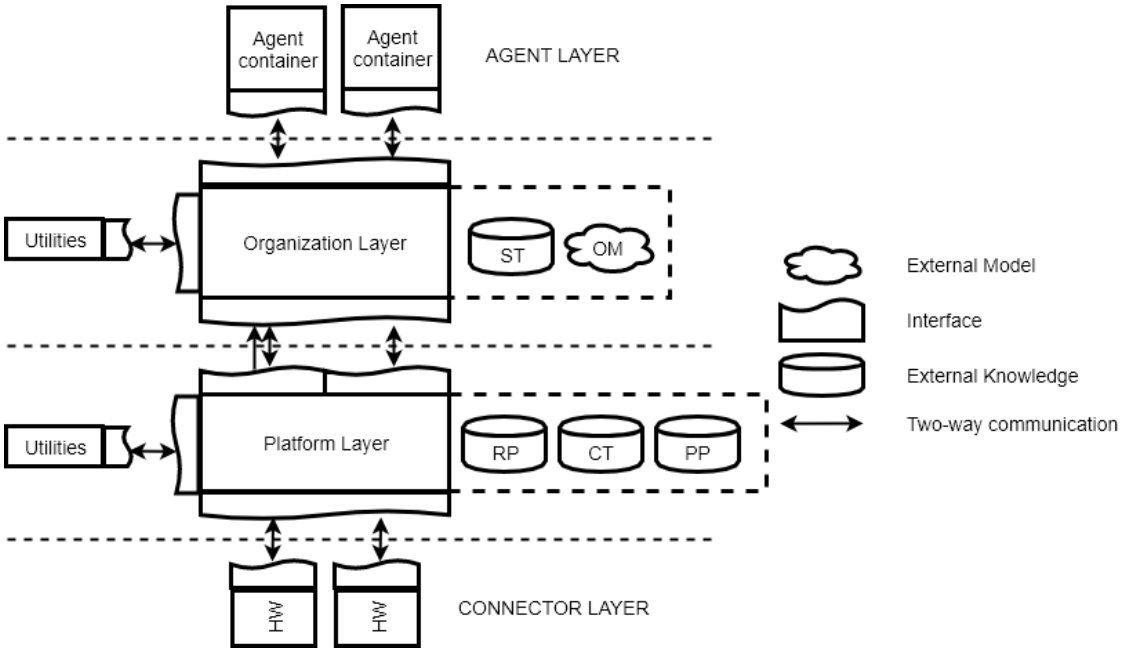

FIGURE 49 : INFRASTRUCTURE MODEL

An important aspect that enable abstraction and high modularity in the system is the usage of interfaces between each layer: if a component of the Agent Layer changes its internal logic but the interface with the Organization Level remains the same, it is still compatible with the lower layers of the system, except for losing compatibility (if not tailored handled) with different Agent Layers technologies if, for example, another structure to pass message between agents is used.

This allows to have different platforms in the network that can implement a different stack but, until the interfaces are the same, different stacks are able to inter-operate. An example is shown in figure 50, where, if a higher layer is missing or exploits another technology, lower layers are able to communicate and provide functionalities to the network.

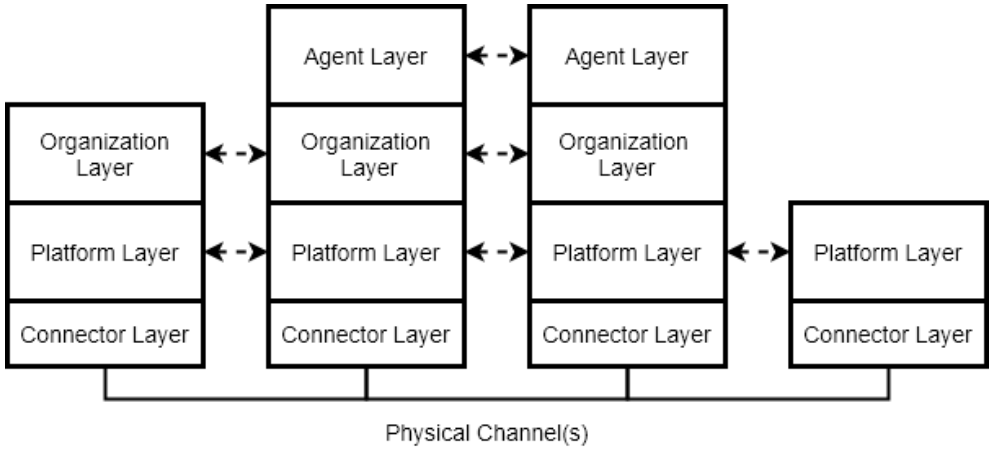

FIGURE 50 : INTERCONNECTION BETWEEN DIFFERENT INFRASTRUCTURES

Some component could be tailored designed to provide only a sub-part of the functionalities of the whole framework (for example, a component composed just by a compatible Platform Layer, could be exploited to act as a communication bridge between two full-stacked infrastructures).

5.2 The Connector Layer

This Layer is composed by different *Connectors* connected to the higher Platform Layer by means of a provided interface. A Connector is a unique media or physical bus that can provide the connection to different RobotParts devices to the Robot. An example of Connector could be the CAN bus, ROS, a web socket, a software used to simulate the movement of the vehicle software or a utility to provide the log of some data. Each connector is defined by a unique name, a description and a version. The Connector Layer is the first and lower level of the infrastructure and act in two different levels:

- **Hardware level**: manage the life cycle of the underlining hardware;
- **Connector level**: manages the connection of the Connector to the Platform layer and the related RobotParts.

5.3 The Platform Layer

The Platform Layer provides the first level of abstraction, providing interfaces for the higher Organization Layer to the RobotPartInterface and to a communication channel by exploiting the RobotParts connected through Connectors provided by

the Connector Layer. The Platform Level is the second level of the infrastructure and act in different levels:

- **Connector level**: manages the different Connector connected to the system by means of a proper interface
- **RobotPart level:** maintains the RobotPartInterface by collecting and organizing the RobotParts from each connected Connector while maintains a database of known RobotPart models that are compatible with the platform and;
- **Workspace level:** provides abstraction of the local and remote Workspace;
- **Communication level**: provides access to a unified communication channels (exploiting different RobotCommunicatingParts) in order to allow a unique transparent communication channel, provides functions to keep track and maintain the topology of a network of platforms and provides abstraction of the remote Workspaces of other platforms deployed in the network;
- **Utility level**: provides functions to monitor and manage the platform.

A model of this layer is shown in figure 51:

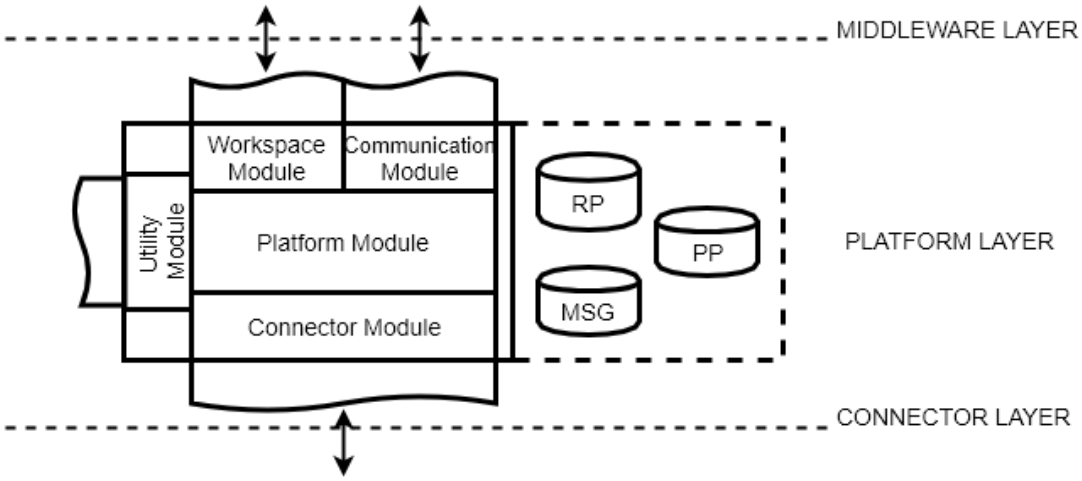

FIGURE 51 : THE PLATFORM LAYER MODEL

In the model it is possible to notice that this layer is composed by five different modules (Platform, Connector, Workspace, Communication and Utility), four of them implement an interface for other software components (Connectors of the

Connector Layer, the Organization Layer and a set of external utilities that can exploit some provided functions in order to show the internal state of the platform and two external databases: one related to compatible RobotParts (RP) and the other regards available Custom Types (MSG). Another file is the Platform Properties file (PP) that defines properties of the Platform instance such as name and version.

5.3.1 Remote Messages Model

A RemoteMessage is a message transmitted between Platforms. The receiver of the message could be the Platform Layer or the Organization Layer of the local platform or another platform. In this last case, the Platform Layer must be able to forward to the target platform, if possible. A Remote Message is composed by the following fields:

- **ReceiverPlatform**: the name of the platform that will receive the message;
- **SenderPlatform**: the name of the platform that sends the message;
- **ReceiverMailbox**: the name of the mailbox that will receive the message.
- **SenderMailbox**: the name of the mailbox that sends the message;
- **Type**: the type of message. The type can be:
    - **Forward**: this type identifies a message that must be received from a higher layer and must be forwarded to the correct Mailbox.
    - **Query:** this type identifies a request from a remote platform.
    - **Response:** this type identifies the response from a remote platform.
    - **Event:** this type identifies a particular event that happened in the network. In this case, the Content field has a named value *eventType* with different values:
        - **NEWPLATFORM:** A platform has been discovered in the network;
        - **DELPLATFORM:** A platform has been removed from the network;
- **Content**: a map of named values. This is the actual payload of the message.

A Mailbox identifies a node where message can be sent or received. A Platform can have different Mailboxes, identified by a unique name, operative at the same time. If

the message is directed to the Platform Layer, the ReceiverMailbox and SenderMailbox fields are empty.

Query and Response message refers to the same subset of message that can be exchanged between two different Platform Layers. The type of message is bundled in the Content field through the "type" named value. When a Platform receives a Query message, the receiver elaborates the requested data and then replies with a Response message to the same platform. The possible messages are:

- **Info**: a resume of the actual status of the platform is requested by another platform. The following information are bundled: platform name, version, robot parts and mailboxes.
- **Remote**: a request of the access to a Robot Part is requested. As content of the message there is the link of the topic or the service of the robot part to be queried.

The usage of Query and Response message allows to have connectionless communication between different platforms.

5.3.2 Modules

As already shown in figure 51, the Platform Layer is structured in 5 modules, with different relevance and duties. It is important to remark that this subdivision is not mandatory but just a reference for the further implementation and to ease the logic division of software components. In fact, until the interfaces are not modified, the internal software can be arranged differently.

The **Platform Module** can be considered the central module of the Platform. It has the duties to:

- Maintain the *RobotPartInterface* (the group of all the RobotParts actually connected to the robot) by managing:
    - the creation and removal of RobotParts from the system
    - the coherence between requests from the Connector Module about the class of RobotParts to be created

- Manage a set of *Properties* of the SP. These define a set of characteristics that can be shared among different platforms in order to be able to recognize themselves. At the moment, the following properties are included:
    - **UUID (Universally Unique Identifier):** a string defining a unique identifier of the platform;
    - **Name:** a string defining the name of the platform;
    - **Description:** a string defining a brief description of the purpose of the platform;
    - **Version:** a string identifying the version and type of the platform, useful to manage interoperation with different versions of platforms. For example, will allow usage of memory constraint devices (such as microcontrollers) that will have a tailored version of the platform, which can be able, however, to communicate with other platforms.
- Interpret and store the database of available RobotParts and Custom Types.

The **Connector Module** manages the different Connectors connected to the system. It has the duties to:

- Creation and maintenance of different Connectors;
- Control if the sensorial data received to from RobotSensingPart and the actuating data that should be sent to a RobotProcessingParts are coherent with the RobotPart model provided during its creation;
- Provide the ConnectorService interface that allows to a lower layer to:
    - Create a new connector;
    - Create/Use/Remove a RobotParts from the connector;
    - Remove the connector from the platform.

The **Communication Module** exploits one or more RobotCommunicatingPart provided from the Platform Module to provide communication functionalities to the higher levels. This module has the duties to:

- Manage the different RobotCommunicatingParts connected to the system;
    - Share the local RobotPartInterface to the other platforms in the system. There are two level of compliance of this property:
        - **Level 0:** No sharing and acquisition of remote RobotPartInterface;

        - **Level 1:** Sharing and acquisition of remote RobotPartInterface. This feature allows, for example, to have a remotely controlled Robot by using a communication channel to pilot it by remotely acting and sensing;
- Manage different *Mailboxes*.
- Manage *Messages* incoming from other Platforms of sent from the local Organization Layer to other Platforms.
- Provide the *CommunicationService* interface able to provide to a higher layer:
    - Create Mailboxes
    - Send/receive message through a Mailbox
    - Remove Mailboxes

There are four different levels of compliance of this module:

- **Level 0:** No RobotCommunicatingPart managed. With this level, only local resources are accessible, no communication is possible with other platforms;
- **Level 1:** One RobotCommunicatingPart managed. With this level, it is possible to have a unique network type, and all platforms, to communicate, must be connected through the same one;
- **Level 2:** Managing of more, separated RobotCommunicatingPart, with this level, it is possible to have more communication channels, each one with a separated list of connected platforms that can be reached through the channel;
- **Level 3:** Managing of more, connected, RobotCommunicatingPart; with this level, it is possible to have more communication channels, which one with a separated list of connected platforms that can be reached. This is the most complex level. Each module acts as a router that is able to take track of different platform on different channels

The **Workspace Module** has the duty to manage the *Workspace* of the platform. The Workspace is a subset of the RobotPartInterface where the RobotCommunicatingParts are removed. It represents all the RobotParts that can be actively used by the higher layers to interact with. With this module, an external component can register to a topic of a RobotSensingPart or of a RobotActuatingPart or call a process of a RobotProcessingPart.

The process of registration to a topic is similar to the common Publisher/Subscriber pattern, where the first entity does not program to send a message to a specific receiver (subscriber), but instead categorize published messages into classes without knowledge if there is any subscriber. In a similar way, subscribers express interest in a class of message, without knowing if there is any publisher that send messages.

The topic or process (called resource) is expressed in a unique way by means of a Link. A link is a string defined as:

Platform:RobotPartName/Resource

Where Platform is the platform where the Resource will be searched, RobotPartName the name of the RobotPart and finally the Resource the name of the Topic or Process of the RobotPart. If the Resource is on the local platform, the reserved platform name "local" is used. The division by means of platform is useful in the case that similar robots (with analogous RobotParts) are deployed in the system. This module has the duties to:

- Manage the local and remote Workspaces;
- Manage the publisher/subscriber logic of the different RobotParts
- Manage the *WorkspaceService* interface that allows to a higher layer to:
    - List links of local and remote Workspaces;
    - Allow to a higher layer to subscribe to a topic/process through a link;
    - Allow to a higher level to be notified when a new topic/process link is added or removed

The **Utility Module** provides some functions useful to access to some data and functionalities of the Platform. This information could be used to develop some accessory utility (such as a web server) that is able to show some information. Its only duty is to manage a PlatformUtility interface able to:

- Print the Properties of the local and remote platforms;
- Print the list of local connectors;
- Print the list of local mailboxes;
- Print the local RobotPartInterface;
- Allow to act or sense a RobotPart of the local RobotPartInterface

- Print a list of events that happened in the Platform;
- Print the content of the local RobotPart database;
- Print information related to other connected platform (such as properties, connectors, RobotPartInterface and mailboxes).

In order to be easily elaborated, it is advised to use a structured human-readable format such as JSON.

5.4 The Organization Layer

The Organization Level is the third level of the infrastructure and the main tasks are:

- Coordinate the operations of different Organization Layer through the exploitation of a common Arbiter
- Share and follow an Organization Model;
- Follow the processing of the loaded Mission
- Assign Roles to available Agents;
- Keep updated the shared environment.

Roles that are played in the same Organization Layer creates a so-called Group of Roles(GoR). Each Organization can have only one GoR.

As it is possible to notice, Agents are considered as resources of the Organization Layer, which can be assigned to a specific role. The outline of the modules of this layer is shown in figure 52, where it is possible to notice five different modules, one Knowledge Database and one External model. The Modules are the Coordination Manager, the Organization Manager, the Environment Manager, the Agent Container Manager and the Organization Utility. The Knowledge Database is the Knowledge Tree, that stores a tree structure used to map RobotPart to compatible Roles, while the External model is the Organizational Model, that express three fundamental aspects of the organization: the Environment, the Roles and the Mission.

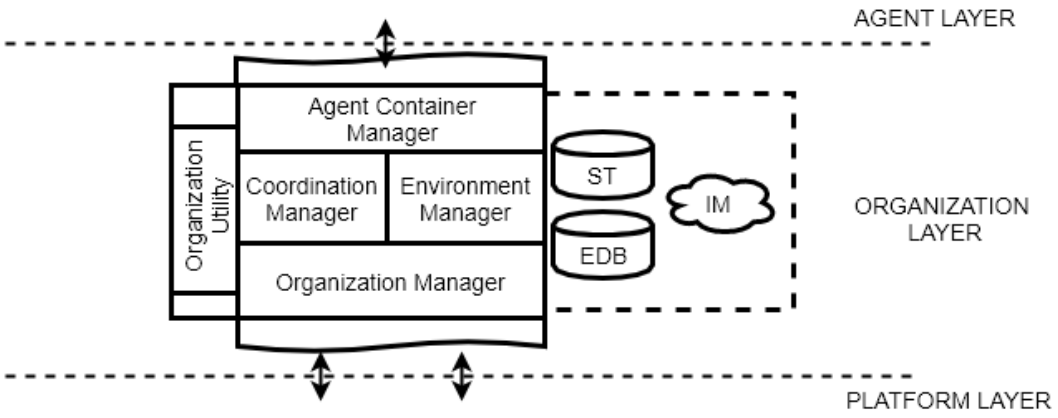

FIGURE 52 : ORGANIZATION LAYER

5.4.1 Modules

The **Coordination Manager Module** manages the cooperation and coordination of different Organization Layers in order to perform joint operations. The coordination is performed by a special Organization Layer that acts as an **Arbiter**, that decides the shared model and coordinate the role assignment. Two scenarios are proposed:

- **Static coordination**: in this scenario, the role of Arbiter is static and related to a specific instance of an Organization Layer. This approach allows to simplify the communication procedures between Organization Layers but can suffer from bottlenecks and single point of failure.
- **Dynamic coordination**: in this scenario, the role of arbiter is dynamically assigned to one Organization Layer. This approach requires proper contract protocols to allow the joint decision of the Arbiter, and the dynamic modification of the shape of the organization.

More specifically, this module is able to:

- Coordinate with other Organization Layers differently in two different cases:
  - In case of dynamic organization:
    - Choose an Organization Layer instance that will act as Arbiter through a contracting procedure;
    - Perform and reply to Arbiter requests;
  - In case of static organization:
    - Register to the local Arbiter or advertise to be the local Arbiter;

        - Perform and reply to Arbiter requests.
- Manage the flow of information received from and transmitted to the Platform Layer to the other modules of the Organization Layer:
    - Addition/removal of RobotParts;
    - Sense/Act from RobotParts associated to agents;
    - Updates to/from the Environments;
    - Messages directed to running agents;
    - Messages directed to the local Organization;
- Manages some information about the actual organization:
    - Actual local and remote assigned Vacant Roles;
    - Status of the actual mission;
- Manage the execution of the Joint Task;

An important figure in the Organization Layer is the Arbiter, a specific Organization Layer with some tailored responsibilities:

- Selection and sharing of a common Organization model;
- Valuation of local and remote Role proposal;
- Sharing of the actual status of the ongoing Mission and of the organization,

In a Dynamic organization, it is not mandatory that all the running Organization Layer are able to acquire the Role of Arbiter. For example, memory constraint devices (such as microcontrollers) will probably refuse to acquire this responsibility.

The **Organization Manager Module** manages the assignment and reasoning of Roles to local agents. More specifically, this module is able to:

- Valuate Vacant Roles which requirements are currently met by the local infrastructure and propose its assignment to an available Agent. The assignment proposal must be accepted from the local Arbiter by sending a request to the local Coordination Manager Module.
- Signal the loss of requirements of a running Role to the local Arbiter by sending a request to the local Coordination Manager Module;
- Request to start and stop agents assigned to roles to a local Agent Container;
- Receive information and requests from the Coordination Layer Module:

- New RobotPart added/removed;
- New list of Vacant Roles available;
- Request to start/stop a role;
- Request to compute the reasoning;

When a particular event occurs from the Coordination Layer Module, this module acts a reasoning process in order to evaluate if it is necessary to take some kind of action. Each action must be subsequently accepted by the local Arbiter. The reasoner takes as input:

- The type of event that occurred;
- The actual Vacant Roles of the Organizational model from the Coordination Manager;
- The Skill Tree model;
- The available Roles from the different connected Agent Containers;

And proposes a list of applicable Roles to the local Coordination Manager, that will forward it to the local Arbiter. Events can be defined for different aspects such as:

- Addition/removal of RobotPart;
- Addition/ removal of available Agents;
- Addition/ removal of applied Role;
- Modification of the actual Organization Model.

The **Environment Manager Module** keeps updated the shared environment, expressed in the Organization model, and the actual status of RobotParts used by Agents. The overall requirement of this module is to keep updated the Environment. The Environment is composed by two different scopes of visibility:

- **Local Environment**: this component is considered local to the single infrastructure instance and contains the actual values of connected RobotParts. These statuses are used by Agents to modify their functioning;
- **Global Environment**: this component is shared and synchronized by the Arbiter between different instances of the infrastructure. It contains information useful to an organizational level: it has the set of

regions (with the related actual resources' value) and the contexts of the running Roles, associated to a region.

The **Agent Container Manager Module** keeps track of different Agent Container connected through the Agent Level and to the associated Agents. These are the main duties:

- Manage the connection of different Agent Containers
- Assign resources to a newly created agent such as:
    - Functions to provide communication with the Organization Manager Module;
    - A communication channel to provide communication to other agents;
    - Access to the shared Environment and to the assigned RobotParts.
    - Provide an *AgentContainerService* interface to allow Agent Layers instances to register and unregister an Agent Container;

The **Organization Utility Module** provides some functions useful to access to some data and functionalities of the Organization Layer. This information could be used to develop some accessory utility (such as a web server) that is able to show some information. Its only duty is to manage a *OrganizationUtility* interface able to:

- Provide information on the loaded Organizational Model and Skill Tree Model;
- Provide information on the connected Agent Container;
- Provide information on the local running Roles and related Agents;
- Provide information about the status of the Environment.

5.5 The Agent Layer

The Agent Layer is the fourth and last level of the proposed infrastructure. There could be more instances of this level on top of the same Organizational Layer. Each instance of this layer possesses these duties:

- Manage, register and unregister an Agent Container by means of the AgentContainerService interface provided by the Organizational Layer.
- Manage a database of possible Agents that can be assigned to proper Roles.
- Manage the life cycle of Agents.

A representation of the Level is shown in figure 53, where it is possible to notice four different modules (Agent Container Module, Agent Executor Module, Agent Factory Module and Agent Utility Module) and one knowledge database, which stores Agent Models (ADB).

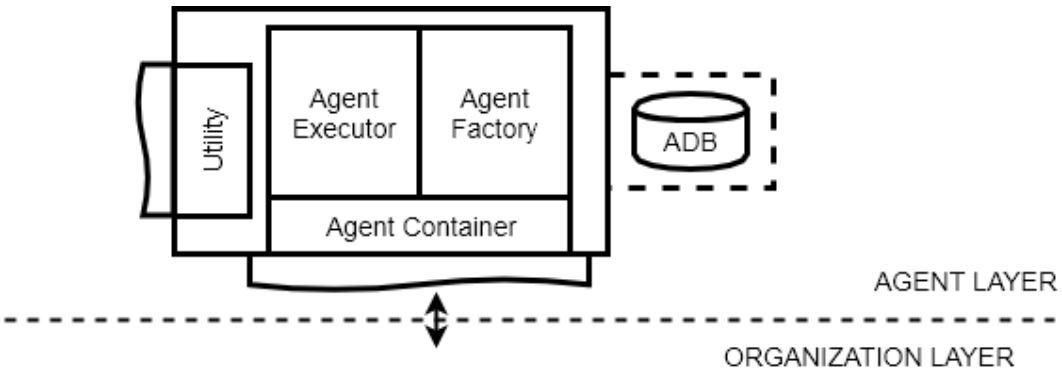

FIGURE 53 : AGENT LAYER

The first module of the Agent Layer is the **Agent Container Module**. This module has the duty to exploit the interface provided by the Organizational Layer to register the Agent Container, and acts as a middleware between the interface and the rest of the Agent Layer, by calling functionalities of other modules.

The **Agent Executor Module**, is the module that manages and monitors the execution of one or more Agents that have a Role assigned. These are the duties of this module:

- Maintain a list of running agents;
- Assign resources at the creation of the agents, following the model of the general Agent showed below;
- Inspect agents, by monitoring their internal stare and functioning;
- Start and stop agents, by requesting them to the Agent Factory Module.

The overall layout of an agent is shown in figure 54.

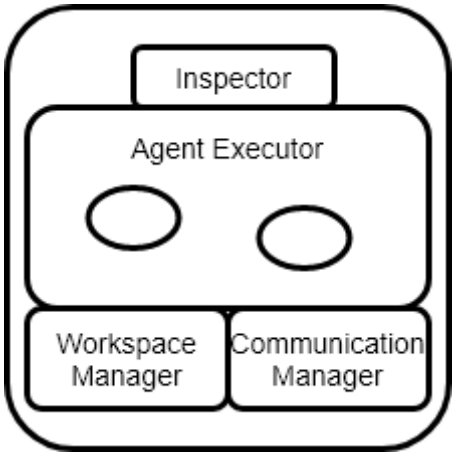

FIGURE 54 : AGENT MODEL

It is possible to define four different components:

- **Agent Executor**: this component manages the life cycle of the agent, which is common for agent from different classes. The life cycle is shown in figure 55. First, the Agent is initialized and reaches the IDLE state. If the agent is cancelled agent state is CANCELLED and then it ends its life cycle. If the agent from the IDLE state is executed the state becomes RUNNING. In this state, the internal logic is executed repeatedly until the agent is stopped, reaching the END state, or paused reaching the IDLE state.

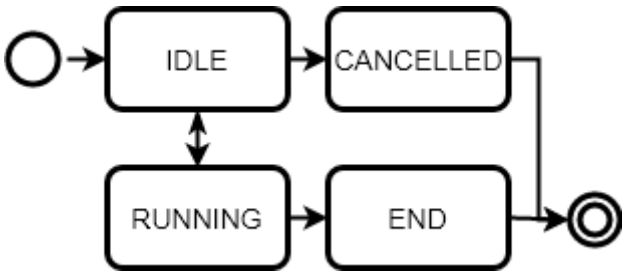

FIGURE 55 : AGENT LIFE CYCLE

- **Communication Manager**: this component takes care of the communication channel of the agent to allow reception of incoming messages from other agents;
- **Workspace Manager**: this component allows the Agent to access to the associated RobotParts;

- **Inspector**: this component allows to inspect and monitor the current state of the agent.

The **Agent Factory Module** is able to read and store internally an Agent Model Database in order to create an Agent instance requested from the Agent Execution Module when needed.

Finally, the **Agent Utility Module** provides some functions useful to access to some data and functionalities of the Agent Layer. This information could be used to develop some accessory utility (such as a web server) that is able to show some information. Its only duty is to manage a *AgentUtility* interface able to:

- List current available and running agents, by showing information provided by the agent's Inspector component;
- Load/unload new Agent Model instances.

5.5.1 Agent Models

The models formally designed (but it is possible to develop other kind of agents) are the following:

- **Automata:** its functioning is based on strict rules called Behaviour;
- **Reactive**: its functioning is based on the execution of a Behaviour Tree (BT);
- **Hybrid**: its functioning is based on a two-tiered vertical architecture.

5.5.1.1 Automata model

In this type of model, the functioning of the agent is highly programmed following a set of rules called *Behaviour*. The agent has different Behaviour that are active at the same time and they are able interact with the environment and other agents. The structure of the functioning of behaviours are like those defined in [118].

These are the common rules for Behaviours:

- It is defined by a unique *Name*, a *Status,* a *Type* and the *Execution Code*;
- Its status can be *Enabled* or *Disabled*; By default, Behaviours are disabled;
- A Behaviours can be enabled at the starting phase of an Agent or in the execution phase by other Behaviours;
- A common life cycle, shown in figure 56. After the starting phase, where the Behaviour is loaded in the *Load* phase, it remains in the *Idle* phase, where it

remains disabled. When the Behaviour is enabled it reaches *Running* state after its initialization in the *Start* phase. When the behaviour finishes its operations, the finalization is carried out in the *End* phase, then the behaviour is removed.

FIGURE 56 : BEHAVIOUR LIFE CYCLE

There are different types of Behaviour:

- **SingleShotBehaviour**: a Behaviour that is executed immediately, one time;
- **CountdownBehaviour**: a Behaviour that is executed delayed, one time. Can be stopped before execution of the code;
- **CyclicBehaviour** a Behaviour that is executed repeatedly, with a defined period. The period can be changed during runtime;
- **EventBehaviour**: this type of behaviour is executed each time a bounding event occurs such as:
    - New message received from another agent;
    - New Sense data received.
- **SFMBehaviour:** This Behaviour defines different *States*, while one of them is the called the *Running State*. Each time this type of Behaviour is called, the Actual State is processed. Inside the State it is possible to alter the current Actual State.

The Behaviour has tailored functions to access to different features during its execution:

- Possibility to act using one of the assigned RobotParts by accessing to the agent's Workspace Manager;
- Send messages to other agents by name or role by accessing to the agent's Communication Manager;
- Store/load values that could be used for future executions of the same behaviour;
- Start/stop another behaviour.

5.5.1.2 Reactive model

In this type of Agent model, the functioning of the agent is defined by means of a Behaviour Trees (BT). A BT is a mathematical model of plan execution. It has been shown in [124] that this type of structure is able to successfully generalize many other control architectures such as the Brook's Subsumption Architecture [18], that is the most well-known reactive architecture. Each Behaviour Tree has many parameters that can be used to parametrize the execution of the tree.

The execution of a Behaviour Tree starts to the Root Node which sends ticks to its child. A tick is a signal that allows the execution of a child. When the execution of a node is allowed a result is provided. The possible results are:

- *SUCCESS*: if the node has achieved its goal;
- *FAILURE*: if the node failed to achieve its goal;
- *EXECUTION*: if the node has not finished yet.

These are the possible type of nodes:

- **Root Node:** this node is unique for each BT, has no parent and just one child, this is the entry point of the execution of the BT;
- **Control Flow Nodes:** they have one parent and at least one child. It is possible to have different type of Control Flow Nodes:
    - **Selector Node:** this node executes its children with a specified order and returns the same result of the first child that returns SUCCESS or EXECUTION. If all children fail, the node will fail. It acts as a OR operator.
    - **Sequence Node:** this node executes its children with a specified order and returns FAILURE when a child returns FAILURE. It acts as a AND operator.
    - **Parallel Node:** this node executes its children at the same time and returns FAILURE when a child returns FAILURE.
- **Decorator Node:** they have one parent and one child. This node applies a transformation to the output of the connected child. For example:
    - **Repeat:** repeats the child node a certain number of times. Returns SUCCESS when finished.

- **Limit:** controls the maximum number of times that a part of the subtree is executed. This node returns SUCCESS until this limit is reached.
- **Invert:** this node changes the result of the child node from SUCCESS to FAILURE and vice-versa.
- **Print:** prints a message in a log, useful for testing the node. Always returns the return of the child.

- **Task Node:** they have one parent and no child and they store the execution code of the BT. The features of the execution code are similar to the ones provided by a SingleShotBehaviour of an Automata Agent.

An example of Behaviour Tree is shown in figure 57. In the example, the agent is an ASV that must get to a certain position, deploy a ROV and maintain a safe distance from it while taking care of the sending of data to a base station. After the end of the mission, the ROV is recovered and the ASV returns to the base.

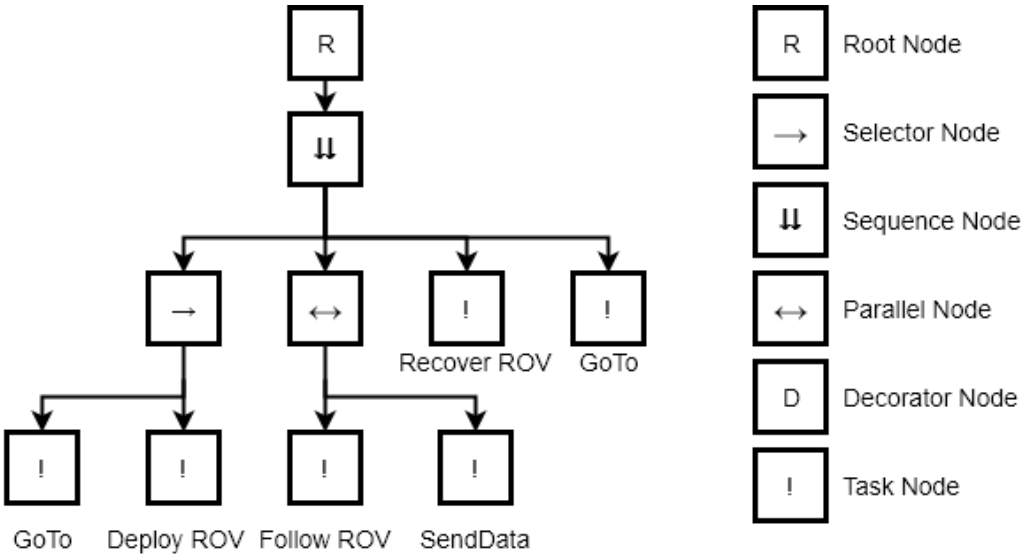

FIGURE 57 : BEHAVIOUR TREE EXAMPLE

5.5.1.3 Hybrid model

In this type of Agent model, the functioning of the agent is defined by a two-tiered horizontal hybrid architecture. A representation of this agent is shown in figure 58.

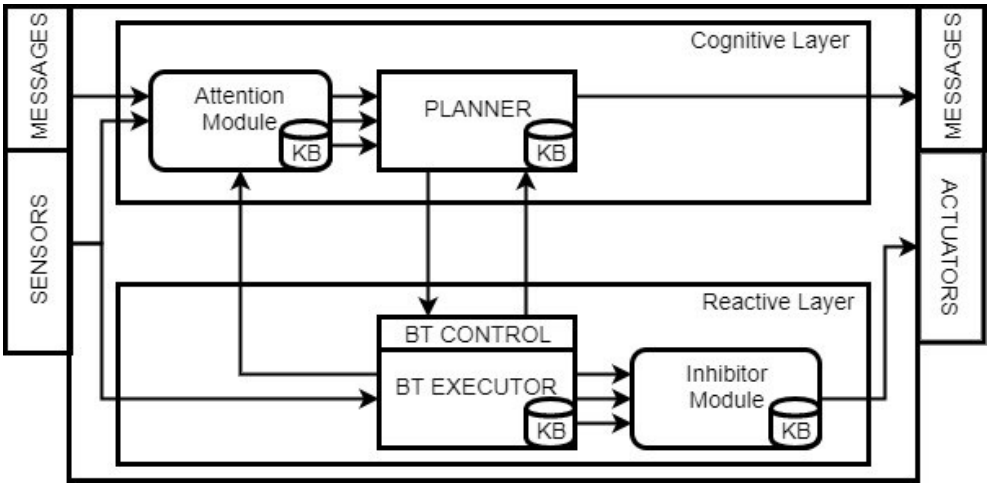

FIGURE 58 : HYBRID AGENT MODEL

It is possible to notice three main components: the input feed, which considers data coming from the Sensors Interface and messages received from other Agents, the concrete agent and the output feed, which considers data flowing through the Actuators Interface and to output messages to other agents.

The concrete agent is characterized by two layers that are called “Reactive” and “Cognitive”. They have their own purposes and communicate in order to provide an internal coordination of the development of each one goals. They follow two different agent models: the reactive model and the BDI model.

It is possible to notice different Knowledge Bases (KB) assigned to different parts of the agents. These KBs are included in a unique Agent Definition that is stored and interpreted by the modules of the Agent Layer.

The *Reactive Layer* acquires directly data feed from the Sensor Component, which is used to provide data to one or more Behaviour Trees that are concurrently executed in the BT Executor component. Their life cycle is maintained by a component called *BT Control*, which is able to communicate with the higher Cognitive layer to start, pause and stop BTs. The BT Executor is called by the BT Control at a fixed rate or when a new sensor data is received to perform a new execution of the BTs. The goal of this layer is to execute correctly the loaded BTs.

It is important to notice that this layer can be connected only to the input and output related to sensors and actuators. In fact, being a reactive component, its reasoning it is based on only the data received from the environment.

The allowed Behaviour Trees are maintained in a proper Knowledge Base. Each BT is defined by the following information:

- A unique name;
- A list of pre-conditions;
- A list of post-conditions;
- The behaviour tree itself, expressed in a similar way to the Reactive Agent model;

The pre- and post- conditions are, respectively, facts that must be valid in order to execute the BT and facts that will be added in the Cognitive Layer when the BT will finish its operations successfully. They are evaluated by the Cognitive layer and embedded in the definition of the BT. This layer has no duty to know what these statements mean.

The generated outputs are collected by a filtering module called Inhibitor Module that will inhibit some of the overlapping outputs (if any) before sending them to the Actuator Component. An output is overlapping with another one if they try to control at the same time the same Actuator. This module evaluates different rules stored in a proper KB.

The *Cognitive Layer*, by the other way, collects filtered information from the Sensor feeds and incoming messages from other agents. This data is gathered in a filtering module called Attention Module, which can be considered a rule evaluator, like the Inhibitor Module. The different Filter Rules are contained in a proper KB. These rules are able to modify and generate sets of Beliefs (sentences that the Agent considers as true) that are used in the Planner Component. Another input to this module are the post-conditions from the BT Executor Module. The Planner Component is motivated as a BDI agent. The Beliefs are provided by the Attention Module and stored in a proper structure. The fact that the structure is permanent but modifiable from the external or the agent itself (representing a sort of memory) differs this layer to the Reactive one that acts in relation to just the real-time external input. The enabled Desires are a subset of the Goals related to the Role from the Organization Model that the agent is currently playing. The KB of this module is represented by a list of Plans, which are a set of action that the agent can do in order

to achieve one or more if its intentions. Tailored functions in the Plans allows to send message to other agents or to communicate with the Reactive Layer to start or stop a Behaviour Tree.

Chapter 6: Development of the infrastructure

In this chapter, two topics will be discussed:

- A first concrete implementation of the infrastructure (models and middleware). For the middleware the requirements and some structural technologies will be introduced; then the implementation of each layer will be briefly presented.
- The description and implementation of a simulation scenario used as a proof of concept to test the functioning of the infrastructure.

6.1 First implementation

In this section, the first implementation of the infrastructure will be shown. This implementation has the following objectives:

- Implement and test Models and their reasonings;
- Show the functioning of a prototype of the middleware;
- Show the implementation of the models related to two types of agents (Automata and Reactive).

6.1.1 Models

Related to the three models (Robot Part model, Skill Tree model and Organization Model) there is no much to add. This information is stored in different text files as JSON [125] (JavaScript Object Notation) objects and structures. JSON is a simple object serialization approach that has been selected for different positive aspects:

- Human readable format;
- Coincide format thanks to named/value pair-based approach;
- High number of libraries in different languages to automatically serialize and de-serialize JSON structures;

- Possibility to write a validations schema, but there is no formal grammar definition such as XML schemas or DTD.

6.1.2 Middleware

6.1.2.1 Employed technologies

To implement the middleware, the language Java has been chosen, together with *OSGi* (Open Service Gateway initiative) and *Apache Karaf* technologies.

OSGi is a framework that usually runs on top of the Java Virtual Machine able help the development and deployment of complex, modular software programs and libraries. As shown in figure 59 the framework is structured in four layers:

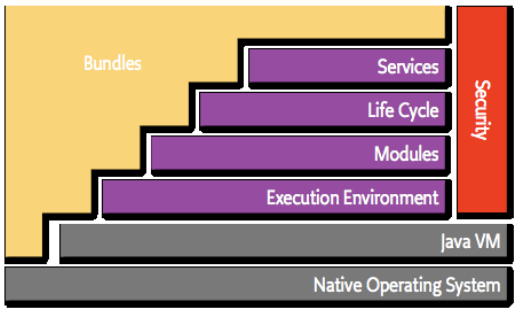

FIGURE 59 : OSGI STRUCTURE

- **L0**: Execution Environment: defines which methods and classes are available for a specific platform;
- **L1**: Modules: an essential feature of OSGi is modularity. This layer allows to define encapsulation and declaration of dependencies between Bundles;
- **L2**: Life Cycle Management: manages the Bundle life cycle without requesting the restart of the VM. This allows to be able to add, remove and update Bundle without compromising the entire system;
- **L3**: Service Registry: predisposes function to allow bundle to communicate between each other's with a publish-find-bind model.

A *Bundle* is essentially a group of Java classes and additional resources made by external developers equipped with a detailed Manifest file. The Manifest file is able to declare some important feature of the Bundle such as the name, version, the activator class (the access point to the Bundle) and imported and exported packaged. In fact, OSGi adds a package-wise visibility in the bundles, allowing to

other bundles to import packages, usually representing interfaces to the functionalities of the Bundle.

Apache Karaf is a lightweight, powerful and enterprise-ready container that implements usually Apache Felix as OSGi frameworks and it has been used as container of the whole infrastructure. Some additional features are advanced log capabilities, definition of property files in order to provide a configuration of the bundle system-wise, a command line to manage the life-cycle of the Bundles and additional classes to allow the definition of new console commands. This latter is useful to exploit the Utility services provided by the different layers of the middleware.

6.1.2.2 Middleware implementation

These are the features of the developed middleware:

- Complete parsing and interpretation of the three models;
- Arbiter implemented as static arbiter;
- Allow the connection of different Connectors;
- Allow the connection of different Agent Containers;
- Managing of one unique communication channel (Level 1 compliance);
- No managing of dynamic topology;
- No managing of remote Robot Parts;

Using OSGi, the different layers are implemented as separated bundles, deployed together inside Karaf following the schema of figure 49. The interfaces between layers shown in middleware structure are developed exploiting OSGi services and dynamic binding capabilities. This allows, for example, to dynamically unregister and register again Connectors and Robot Parts when the Platform component is updated, of to automatically register a new Agent Container when a new Agent Layer is deployed in the system.

About the agents, they are also stored in a JSON file that includes the Behaviours of an Automata Agent and the Behaviour Trees of a Reactive Agent, incorporating the information showed in 5.5.1.1 and 5.5.1.2 . The agent Factory module can read these files and to create models of agents that can be created when requested from the Agent Executor module. Each agent follows the structure shown in figure 54. The

executing part of the Behaviours and of the Task nodes are performed by means of Lua scripts embedded in the structure that is processed at execution time by the Java code. The script has access to a set functions to send messages to other agents, act, sense the environment, store/recall values in memory and so on. In the meantime, different sources of information (such as parameters of the agent and of the Behaviour or Behaviour Tree), are transferred to the script by means of Lua tables, that can be directly accessed from the script. This allows to add, modify or remove the agent logic code while the system is running without compromising it. In case of a Task node of a Behaviour Tree, it is requested to return the state of its execution (SUCCESS, FAILED or PROCESSING).

6.2 Test of the concrete implementation: model of an ASV system for autonomous navigation

6.2.1 Scenario

In this scenario, a simulation of the functioning of the ASV shown in paragraph 3.4 will be implemented. First of all, the three models will be introduced and then the actual implementation of the connector and of the agent functioning.

Reassuming, the scenario regards the modelling of two entities, an ASV and a Ground Station. The ASV is composed by different components: a processing board that stores the infrastructure and communicates with the other components through a unique bus, a rudder that is able to allow the steering to the vehicle, a thruster that is able to provide speed to the vehicle, an IMU able to provide orientation information and the GPS that can provide the actual position of the vehicle. The Ground Station is provided with a GUI to allow the monitoring of the ASV and to provide the reference GPS position to the ASV.

The mission of the agency is to allow the automatic navigation to the ASV by reaching the reference point provided by the Ground Station.

6.2.2 Models

6.2.2.1 Robot Part Model

In the model are defined four different RobotParts:

- **Thruster**: represents the electric offboard engine. It is an Actuating RobotPart and its only Topic is called “speed”. This topic has just one FLOAT parameter called value.
- **Rudder**: represents the rudder that is allow the boat to steer. It is an Actuating RobotPart and its only Topic is called “angle”. This topic has just one FLOAT parameter called value.
- **Gps:** represent a device able to provide the position data for the Robot. Is s considered a Sensing RobotPart and its only topic is “position”. This topic is composed by two FLOAT parameters: *latitude* and *longitude*.
- **Imu**: represent a device able to provide the orientation data for the Robot. It is considered a Sensing RobotPart and its only topic is “orientation”. This topic is composed by three FLOAT parameters: *pitch*, *roll* and *yaw*.

6.2.2.2 Skill Tree Model

The Skill Tree Model of the simulation is shown in figure 60:

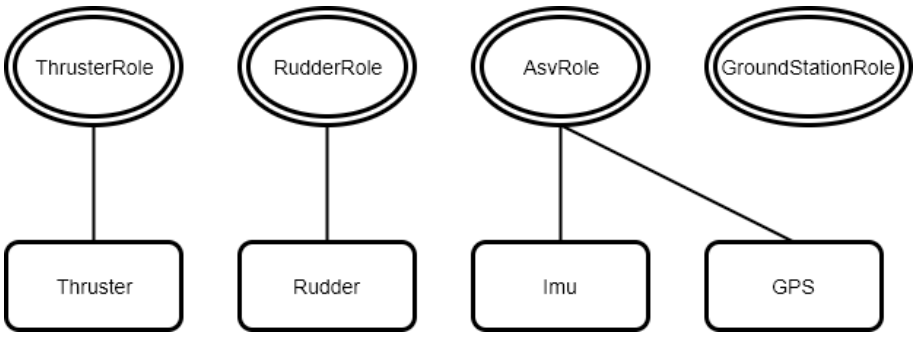

FIGURE 60 : SKILL TREE MODEL OF THE SIMULATION

It is possible to notice that there are four decorated roles:

- **ThrusterRole & RudderRole:** these roles have one decorated RobotPart child each: thruster and rudder, respectively;
- **AsvRole:** this role has two decorated RobotPart children: imu and gps.
- **GroundStationRole:** this role has no child, if an Agent Container proposes this role as available, it will be allocated without any kind of constraint.

6.2.2.3 Organization Model

The Organization Model is the following:

- **Environment**: two regions are modelled: Marine and Land. The ASV will be inserted in the first environment, while the Ground Station in the second one. In this scenario, no Global or Local resources are inserted.
- **Roles**: there are 4 roles in the organization:
    - **ThrusterRole:** the Role Context of the role is composed by a single FLOAT argument called Speed, and there is only a behaviour called setSpeed that receives as parameter the speed to set. There are no Goals, Tasks or Super Roles.
    - **RudderRole:** the Role Context of the role is composed by a single FLOAT argument called Angle, and there is only a behaviour called setAngle that receives as parameter the angle to set. There are no Goals, Tasks or Super Roles.
    - **AsvRole:** the Role Context of the role is composed by three FLOATS, and there is only a behaviour called GoTo that receives as parameter the position (defined as latitude and longitude) to reach. There are no Goals, Tasks or Super Roles.
    - **GroundStationRole:** this role has no Role Context, Behaviours, Goals, Tasks or Super Roles.
- **Mission:**
    - **Vacant Roles**: the vacant roles are four, one for each role with a minimum and maximum cardinality of 1.
    - **Joint Task:** in this simple scenario, no joint task is available, because the only interaction between agents is through direct message sending. In this case the Arbiter doesn't know directly when the mission is completed.

6.2.3 Connectors

There is a unique connector that allows to simulate the actuators and sensors of the system called **SimulatedAsvConnector**. This connector is able to generate the four

required RobotParts to allow the functioning of the ASV (Rudder, Thruster, Imu and GPS) and is able to:

- Simulate the movement of the ASV by receiving the speed of the Thruster and angle of the Rudder and generating the simulated orientation and position data;
- Provide a GUI to allow the monitoring of the simulation, to confirm the correct functioning of the system. This simple GUI is shown in figure 61. In the centre, the tracking and the actual position and orientation of the ASV is shown by means of an arrow marker, while the track done is presented in blue. In the bottom of the GUI, some labels show the speed and the angle received from the ASVAgent and actual simulated position and orientation. During the simulation, data related to the actual position, speed of the thruster and angle of the rudder is logged in a tailored text file.

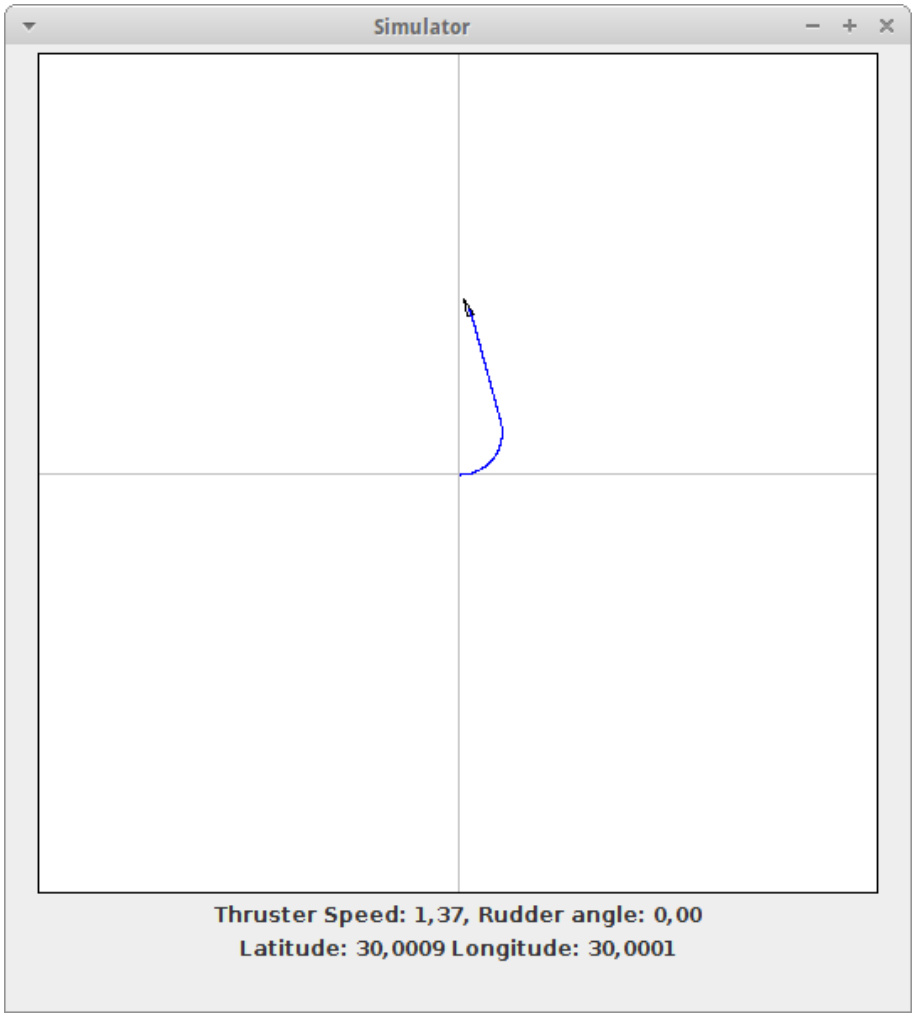

FIGURE 61 : SIMULATEDASVCONNECTOR GUI

6.2.4 Agents

About the agents, there are two different Agent Containers, providing two different set of Agents.

The first one is called ASVContainer and is able to generate and maintain Automata and Reactive agents. The agents that are included are:

- **ThrusterAgent and RudderAgent**: these agents are two Automata agents that can play the respectively the ThrusterRole and RudderRole roles. They are composed by a single MessageEventBehavior that allows to receive the information related to speed of the thruster and the angle of the rudder, filter it and send an act command to the respective RobotParts;
- **ASVAgent**: this agent is implemented as a Reactive agent and can play the ASVRole with a Behaviour Tree called "Navigation" that receives, as parameters, two floats that identify target latitude and longitude. The BT is shown in figure 62. The Behaviour Tree implements the following logic through a sequence node: first of all, in the Calculate Task Node the bearing and the Distance between the ASV and the target are calculated and stored. Then in the Align Task Node the correct alignment with the target is checked and performed if the ASV is misaligned by sending messages to the agents that play the ThrusterAgent and RudderAgent roles and returning PROCESSING as state. When the alignment is performed, the state return SUCCESS and the Proceed Task node is executed. This node assigns a speed to the thruster proportionate to the distance from the target and resets the angle of the rudder. If the distance from the target is less than a fixed value the ASV keeps the position by stopping the thruster. If during the Proceed phase the ASV loses its alignment, the sequence allows to give predominance to the Align Task Node.

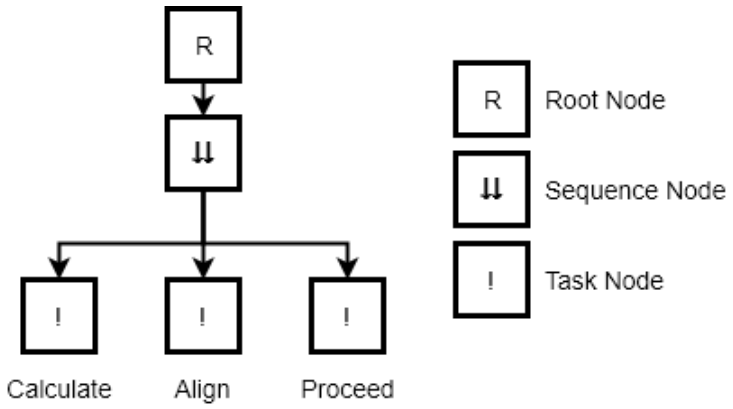

FIGURE 62 : BEHAVIOUR TREE OF THE ASVAGENT

The second container is called GroundStationContainer and is able to generate one type of custom agent, named GSAgent, which is able to play the GroundStationRole role. This role is provided with a simple GUI (shown in figure 63) which allows to monitor the status of the underlining agent, to send a message to the ASVAgent concerning the GPS coordinates to acquire and to show the status of the shared environment.

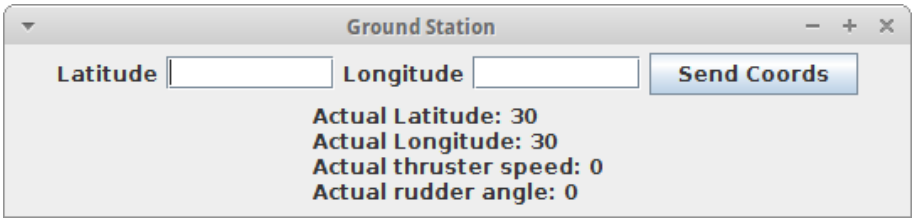

FIGURE 63: GROUND STATION GUI

6.2.5 Simulations

Some simulations were performed by exploiting a single Karaf instance and deploying the developed middleware, the simulated ASV Connector and the two Agent Containers, one for the Agents related to the ASV, and one for the Agent of the Ground Station.

At the start of the infrastructure the RobotParts are correctly registered to the system, the models loaded and the Agent Containers are ready to start new agents. When the Organizational Model is loaded and parsed, the four agents are requested, with the correct RobotParts associated to them.

An external Bundle is able to communicate through the Utility Modules of different Layers of the Middleware and to provide command through the Karaf command

interface to monitor the current values in the Environment, the status of the running roles and the status of the connected RobotParts.

With the GUI of the Ground Station it is possible to set a new target point for the ASVAgent by sending the request of a new behaviour to it. At the reception of the request, a proper Behaviour Tree is created and executed by the ASVAgent, following the Behaviour Tree shown in figure 62. This BT is repeatedly executed until the requested point is reached. In a more complex scenario, this tree could be just one of the running Behaviour Trees, or a sub tree of another one.

One of the simulation result is shown in figure 64. The target position is indicated with the red circle.

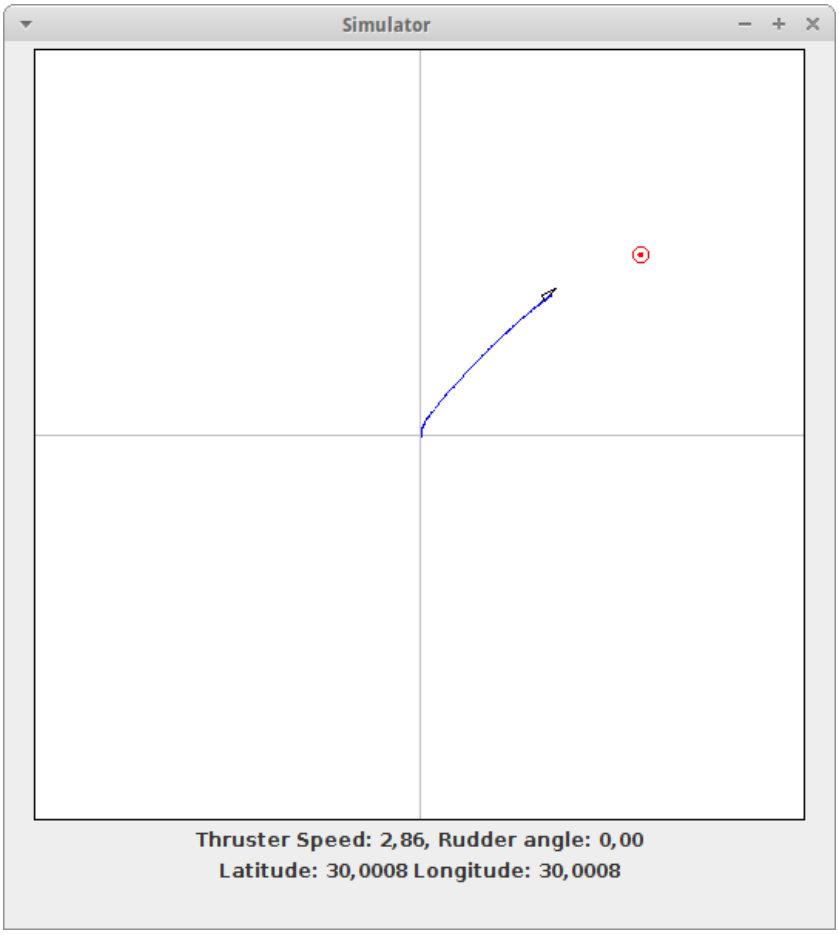

FIGURE 64 : GUI DURING THE SIMULATIONS

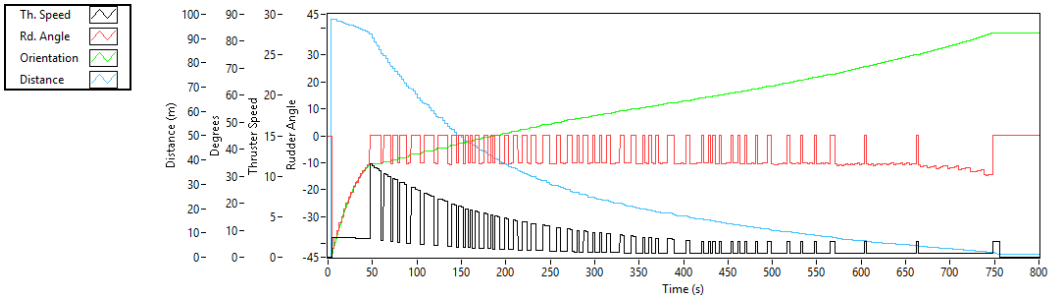

FIGURE 65 : DATA ACQUIRED FROM SIMULATIONS

The results of the simulation shown in figure 65 show that the navigation algorithm implemented through the reactive agent that uses BTs can successful reach the requested GPS position. In the figure it is possible to notice that the distance diminishes until the dead zone value is obtained and the speed of the thruster is zeroed after less than 800s. In the first part of the algorithm the vehicle rapidly aligns and then approaches with the target by regulating the rudder angle and speed in relation to the distance.

Chapter 7: Conclusions

7.1 Thesis review

In Chapter 1 a state of the art of different arguments related to the main topic of this dissertation has been introduced. From the analysis of ontologies for single-robot and multi-robot systems, models of agents, of Multi-Agent Systems characterization and of researches about specifications and features of the environment a solid terminology and knowledge of this field of study was collected. Starting from the technologies presented in Chapter 3 different ideas were taken to shape and design the infrastructure presented in chapter 4 and 5. The *DocuScooter* presented the requirement to express in a formal way different third-party technology that could be attached by means of a model able to express not only the type of device but also the type of data exchanged, in a similar way to the concept of RobotPart introduced in the infrastructure. With *OpenFISH*, inside the laboratory we started to design a first version of multi-agent system divided in two vertical layers: a lower one that could be compared with the Connector Layer of the architecture and a higher one that implements the NGC logic. Another important aspect was the designing of the mission by means of an interpreted language, in the same way the Joint Task and Agents are designed. In the Home Automation System, a focus was put in the formalization of agents and of the resources of environment. The MAS for a general purpose ASV has been another good example of custom, general purpose MAS based on ROS, where the agent were shaped as Automata agents. In Chapter 4 the first component of the infrastructure has been presented: a set of three models able to express the components of the robots, how the robot can use these components and how the robot can coordinate with other robots in a shared environment. In Chapter 5 the second component of the infrastructure is presented: a four-layered vertical middleware model. The model proposes a set of guidelines, requirements and duties of each layer hat can be implemented using different technologies and for different situated robots. It is possible to be compliant up to different degrees for each component and the usage of interfaces enables easy interoperation between

different robots. Finally, Chapter 6 presents a first and partial implementation of the infrastructure using Java and OSGi, by providing a ground to develop a first simulated scenario composed by an ASV with different, autonomous components that must be able to coordinate with a ground station to reach a given position.

7.2 Realization of objectives

The two main objectives of this dissertation have been achieved.

In the first part (Chapter 2 and 3) of the dissertation a wide analysis of the state-of-the art related to robots, MAS and related concepts is presented together with technologies that influenced the following development.

In the second part of the dissertation (Chapter 4, 5 and 6), three models and the layout of the architecture which allows the control of the whole MRS has been introduced, implemented and partially tested in a simulated scenario.

7.3 Future steps

Future improvements on the infrastructure could be considered on three different aspects: models, middleware and implementation.

About the models, a refactoring could be made especially for the Organization Model, when considering more complex and general scenario. For example, a deepened importance should be given to the Environment, implementing, as instance, laws (expressed in logical language) able to manage and control the resource allocation by means of the Arbiter. Up to now, how resources are used and shared are a responsibility embedded by the single agents or the Arbiter, but no standardization is given to how the resource are allocated. Another kind of consideration could be made for the Mission part of the model. At the moment, for example, there is no strict specification and language for the Joint Task.

About the middleware, a refactoring should be conducted to better specifies some aspects of it such as managing of the communication channel with tailored protocols to maintain a dynamical topology of the network through different RobotCommunicatingParts while a study should be conducted about the Arbiter

managing by implementing more robust contracting algorithms to manage Arbiter disconnection and information update in a dynamic organization scenario.

The actual implementation is not at full potential, a lot of component should be refactored and enhanced by following the modifications of the structure of models and middleware such as provide better reasoning by evaluating performance of different agents, better communication and middleware manager modules and implement the Hybrid agent model. Moreover, a lighter version of the middleware could be designed aimed for memory constraint devices, which could be anyway able to communicate with more complex and complete instances of the infrastructure. Finally, an important step will be the actual implementation of the designed technologies in tailored simulation, concrete vehicles by implementing a wire amount of connector in order to have access to a wide number of available hardware.

References

[1] N. Nilsson, “Shakey the robot,” Artificial Intelligence Centre, Menlo Park, 1984.

[2] "IEEE Standard Ontologies for Robotics and Automation," *IEEE Std 1872-2015,* pp. 1-60, 2015.

[3] A. Pease, “Standard Upper Ontology Knowledge Interchange Format,” 2004.

[4] I. Niles and A. Pease, “Towards a Standard Upper Ontology,” *Proceedings of the international conference on Formal Ontology in Information Systems,* 2001.

[5] H.-M. Huang, “Autonomy Levels for Unmanned Systems Framework,” *National Institute of Standards and Technology,,* p. Special Publication 1011, 2004.

[6] A. Farinelli, L. Iocchi and D. Nardi, “Multirobot systems: a classification focused on coordination,” *IEEE Transactions on Systems, Man, and Cybernetics,* vol. 34, no. 5, pp. 2015-2028, 2004.

[7] M. Woolridge, Introduction to Multi Agent Systems, John Wiley & Sons, 2009.

[8] G. Weiss, Multiagent systems (2nd ed.), Cambridge: The MIT Press, 2013.

[9] F. Heylighen, “Stigmergy as a universal coordination mechanism I: Definition and components,” *Cognitive Systems Research,* pp. 4-13, 2016.

[10] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Upper Saddle River, New Jersey: Prentice Hall, 2003.

[11] M. Fisher, “A Knowledge-Theoretic Semantics for Concurrent METATEM,” *Temporal Logic - Proceedings of the First International Conference,* pp. 480-505, 1994.

[12] G. De Giacomo, Y. Lespérance and H. J. Levesque, "ConGolog, a concurrent programming language based on the situation calculus," *Artificial Intelligence,* vol. 121, no. 1, pp. 109-169, 2000.

[13] H. Levesque, R.Reiter, Y. L.Espèrance, F.Lin and R. Scherl, "GOLOG: a logic programming language for dynamic domains," *Journal of Logic Programming,* pp. pp 59-84, 1997.

[14] J. McCarthy and P.Hayes, Some phylosophical problems from the standpoint of artificial intelligence, Edimburgh University Press, 1969.

[15] S. J. Rosenschein, "Formal theories of knowledge in AI and robotics," *New Generation Computing,* pp. 345-357, 1985.

[16] S. J. Rosenschein, "Synthesizing information tracking automata from environment descriptions," *Proceedings of Conference on Principles of Knowledge Representation and Reasoning,* 1989.

[17] L. P. Kaelbling, "Goals as Parallel Program Specifications," *Proceedings of the Seventh National Conference on Artificial Intelligence,* 1988.

[18] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal on Robotics and Automation,* vol. 2, no. 1, pp. 14-23, 1986.

[19] P. E. Agre and D. Chapman, "Pengi: lmpementation of a Theory of Activity," *Proceedings of the 6th National Conference on Artificial Intelligence,* pp. 268-272, 1987.

[20] J. A. Firby, "An investigation into reactive planning in complex domains," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence,* pp. 202-206, 1987.

[21] P. Maes, "The Agent Network Architecture (ANA)," *ACM SIGART Bulletin,* vol. 2, no. 4, pp. 115-120, 1991.

[22] M. Minsky, The Society of Mind, New York: Simon & Schuster, 1986.

[23] D. Dennett, “Intentional Systems Theory,” in *The Oxford Handbook of Philosophy of Mind*, 2009.

[24] M. E. Bratman, Intentions, Plans, and Practical Reason, Cambridge: Harvard University Press, 1987.

[25] M. P. Georgeff and A. L. Lansky, “Reactive Reasoning and Planning,” *AAAI'87 Proceedings of the sixth National conference on Artificial intelligence,* pp. 677-682, 1987.

[26] M. P. Georgeff, A. L. Lansky and M. Schoppers, “Reasoning and planning in Dynamic environments: An expreriment with a Mobile Robot,” Tech.Note 380, Artificial Intelligence Centre, Menlo Park, 1987.

[27] M. P. Georgeff and A. L. Lansky, “A System for Reasoning in Dynamic Domains: Fault Diagnosys on the Space Shuttle,” Tech.Note 375, Artificial Intelligence Center, Menlo Park, 1986.

[28] J. H. Connell, “SSS: a hybrid architecture applied to robot navigation,” *Proceedings 1992 IEEE International Conference on Robotics and Automation,* vol. 3, pp. 2719--2724, 1992.

[29] R. P. Bonasso, D. Kortenkamp, D. P. Miller and M. Slack, “Experiences with an architecture for intelligent, reactive agents,” *Intelligent Agents II Agent Theories, Architectures, and Languages,* pp. 187--202, 1996.

[30] D. P. Miller and M. G. Slack, “Increasing access with a low-cost robotic wheelchair,” *Proceedings of IROS '94,* pp. 1663-1667, 1994.

[31] R. J. Firby, *Adaptive Execution in Complex Dynamic World.,PhD Thesis,* Yale University, 1989.

[32] C. Elsaesser and M. G. Slack, “Integrating deliberative planning in a robot architecture,” *In Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space* , 1994.

[33] R. C. Arkin and T. Balch, "AuRA: Principles and Practice in Review," *Journal of Experimental and Theoretical Artificial Intelligence,* vol. 9, pp. 175-189, 1997.

[34] J. P. Müller and M. Pischel, "The agent architecture InteRRaP : concept and application," DFKI Deutsches Forschungszentrum für Künstliche Intelligenz, 1993.

[35] H. J. Burckert and J. Muller, "RATMAN: Rational Agents Testbed for Multi-Agent Networks," in *Decentralized A.I., 2*, North-Holland, 1990, pp. 217-231.

[36] I. A. Ferguson, Touring Machines: An Architecture for Dynamic, Rational, Mobile Agents, University of Cambridge Computer Laboratory, 1992.

[37] D. Vernon, G. Metta and G. Sandini, "A Survey of Artificial Cognitive Systems: Implications for the Autonomous Development of Mental Capabilities in Computational Agents," *Evolutionary Computation,* vol. 11, no. 2, pp. 151-180, 2007.

[38] S. Profanter, "Cognitive architectures," in *HaupteSeminar Human Robot Interaction*, 2012.

[39] A. V. Samsonovich, "Toward a Unified Catalog of Implemented Cognitive Architectures," *Proceedings of the 2010 conference on Biologically Inspired Cognitive Architectures 2010,* pp. 195-244 , 2010 .

[40] J. Laird, A. Newell and P. Rosenbloom, "SOAR: An architecture for general intelligence," *Artificial Intelligence 33,* pp. pp. 1-64, 1987.

[41] A. Newell, Unified theories of cognition, Cambridge: Harvard University Press, 1990, pp. Harvard University Press, Cambridge, MA.

[42] R. Sun, "The CLARION cognitive architecture: Extending cognitive modeling to social simulation.," in *Cognition and Multi-Agent Interaction*, New York, Cambridge University Press, 2004.

[43] P. Langley, "Cognitive Architectures and the Construction of Intelligent Agents," in *Intelligent Agent Architectures*, Stanford, 2004.

[44] K. P. Sycara, "Multiagent Systems," *AI magazine,* vol. 19, no. 2, pp. 79-92, 1998.

[45] G. Picard, J. F. Hübner, O. Boissier and M.-P. Gleizes, "Reorganisation and Self-organisation in Multi-Agent Systems," *ORGMOD,* pp. 66-80, 2009.

[46] J. Ferber, O. Gutknecht and F. Michel, "From Agents to Organizations: An Organizational View of Multi-agent Systems," *LNCS n. 2935: Procs. of AOSE'03,* pp. 214-230, 2003.

[47] J. Ferber and O. Gutknecht, "A meta-model for the analysis and design of organizations in multi-agent systems," *Proceedings International Conference on Multi Agent Systems,* pp. 128-135, 1998.

[48] 0. Gutknecht and J. Ferber, "Madkit reference page," [Online]. Available: http://www.lirnun.fr/-gutkneco/madkit.

[49] M. Hannoun, O. S. J. S. Boissier, S. J. Simão, E. Nat and S. Mines, Moise: An Organizational Model for Multi-Agent Systems, Berlin Heidelberg: Springer, 2000.

[50] D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet and W. Joosen, "The MACODO Middleware for Context-driven Dynamic Agent Organizations," *ACM Transactions on Autonomous and Adaptive Systems (TAAS),* vol. 5, no. 1, pp. 1-28, 2010.

[51] M. D'Inverno and M. Luck, Understanding Agent Systems, Springer, 2004.

[52] H. F. Wang, "Multi-agent co-ordination for the secondary voltage control in power-system contingencies," *IEE Proceedings - Generation, Transmission and Distribution,* vol. 148, pp. 61-66, Jan 2001.

[53] S. D. J. McArthur, S. M. Strachan and G. Jahn, "The design of a multi-agent transformer condition monitoring system," *IEEE Transactions on Power Systems,* vol. 19, pp. 1845-1852, Nov 2004.

[54] T. Nagata and H. Sasaki, "A multi-agent approach to power system restoration," *IEEE Transactions on Power Systems,* vol. 17, pp. 457-462, May 2002.

[55] Y. Tomita, C. Fukui, H. Kudo, J. Koda and K. Yabe, "A cooperative protection system with an agent model," *IEEE Transactions on Power Delivery,* vol. 13, pp. 1060-1066, Oct 1998.

[56] M. Pipattanasomporn, H. Feroze and S. Rahman, "Multi-agent systems in a distributed smart grid: Design and implementation," in *2009 IEEE/PES Power Systems Conference and Exposition*, 2009.

[57] A. Dimeas and N. Hatziargyriou, "A multiagent system for microgrids," in *IEEE Power Engineering Society General Meeting, 2004.*, 2004.

[58] D. E. Olivares, A. Mehrizi-Sani, A. H. Etemadi, C. A. Cañizares, R. Iravani, M. Kazerani, A. H. Hajimiragha, O. Gomis-Bellmunt, M. Saeedifard, R. Palma-Behnke, G. A. Jiménez-Estévez and N. D. Hatziargyriou, "Trends in Microgrid Control," *IEEE Transactions on Smart Grid,* vol. 5, pp. 1905-1919, July 2014.

[59] A. L. Kulasekera, R. A. R. C. Gopura, K. T. M. U. Hemapala and N. Perera, "A review on multi-agent systems in microgrid applications," in *ISGT2011-India*, 2011.

[60] W. Mansour and K. Jelassi, "Flexible Software solution for intelligent multi-agent manufacturing systems," in *2014 International Conference on Electrical Sciences and Technologies in Maghreb (CISTEM)*, 2014.

[61] C. Yunqing and Z. Guoqiang, "Research on Multi-agent System in Intelligent Manufacturing with Enterprise Alliance," in *2015 International Conference on Industrial Informatics - Computing Technology, Intelligent Technology, Industrial Information Integration*, 2015.

[62] C. Alexakos and A. P. Kalogeras, "Internet of Things integration to a Multi Agent System based manufacturing environment," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, 2015.

[63] N.-F. Xiao and S. Nahavandi, "Multi-agent model for robotic assembly system," in *Proceedings of the 5th Biannual World Automation Congress*, 2002.

[64] G. Conte and D. Scaradozzi, "An Approach to Home Automation by means of MAS Theory," *Modeling and Control of Complex Systems,* pp. 461-485, 2007.

[65] Z. Wang, X. Liu and S. Wu, "BACnet intelligent home supervisory control system based on multi-agent," in *The 7th International Conference on Advanced Communication Technology, 2005, ICACT 2005.*, 2005.

[66] M. Ruta, F. Scioscia, G. Loseto and E. D. Sciascio, "Semantic-Based Resource Discovery and Orchestration in Home and Building Automation: A Multi-Agent Approach," *IEEE Transactions on Industrial Informatics,* vol. 10, pp. 730-741, Feb 2014.

[67] D. Mahmood, N. Javaid and M. Ilahi, "Home Energy Management Using Multi Agent System for Web Based Grids," in *2016 International Conference on Frontiers of Information Technology (FIT)*, 2016.

[68] W. Li, T. Logenthiran, W. L. Woo, V. T. Phan and D. Srinivasan, "Implementation of demand side management of a smart home using multi-agent system," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, 2016.

[69] M. Tornow, A. Al-Hamadi and V. Borrmann, "A multi-agent mobile robot system with environment perception and HMI capabilities," in *2013 IEEE International Conference on Signal and Image Processing Applications*, 2013.

[70] C. C. Sotzing, J. Evans and D. M. Lane, "A Multi-Agent Architecture to Increase Coordination Efficiency in Multi-AUV Operations," in *OCEANS 2007 - Europe*, 2007.

[71] X. Q. Bian, T. Chen, J. Zhou and Z. Yan, "Research of Autonomous Control Based on Multi-Agent for AUV," in *2009 International Workshop on Intelligent Systems and Applications*, 2009.

[72] S. I. Roumeliotis and G. A. Bekey, "Collective localization: a distributed Kalman filter approach to localization of groups of mobile robots," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, 2000.

[73] M. Moarref and H. Sayyaadi, "Facility Location Optimization via Multi-Agent Robotic Systems," in *2008 IEEE International Conference on Networking, Sensing and Control*, 2008.

[74] G. Franck, C. Vincent and C. Francois, "A reactive multi-agent system for localization and tracking in mobile robotics," in *16th IEEE International Conference on Tools with Artificial Intelligence*, 2004.

[75] A. Husain, H. Jones, B. Kannan, U. Wong, T. Pimentel, S. Tang, S. Daftry, S. Huber and W. L. Whittaker, "Mapping planetary caves with an autonomous, heterogeneous robot team," in *2013 IEEE Aerospace Conference*, 2013.

[76] T. Huntsberger, P. Pirjanian, A. Trebi-Ollennu, H. D. Nayar, H. Aghazarian, A. J. Ganino, M. Garrett, S. S. Joshi and P. S. Schenker, "CAMPOUT: a control architecture for tightly coupled coordination of multirobot systems for planetary surface exploration," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans,* vol. 33, pp. 550-559, Sept 2003.

[77] A. Howard, L. E. Parker and G. S. Sukhatme, "Experiments with a Large Heterogeneous Mobile Robot Team: Exploration, Mapping, Deployment and Detection," *The International Journal of Robotics Research,* vol. 25, pp. 431-447, 2006.

[78] M. C. G. Quintero, J. A. O. Lopez and R. F. A. Bertel, "Coordination mechanisms for a multi-agent robotic system applied to search and target location," in *IX Latin American Robotics Symposium and IEEE Colombian Conference on Automatic Control, 2011 IEEE*, 2011.

[79] A. Marjovi, J. G. Nunes, L. Marques and A. de Almeida, "Multi-robot exploration and fire searching," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[80] S. Mendes, J. Queiroz and P. Leitão, "Data driven multi-agent m-health system to characterize the daily activities of elderly people," in *2017 12th Iberian Conference on Information Systems and Technologies (CISTI)*, 2017.

[81] G. Itabashi, M. Chiba, K. Takahashi and Y. Kato, "A Support System for Home Care Service Based on Multi-agent System," in *2005 5th International Conference on Information Communications Signal Processing*, 2005.

[82] R. C. Arkin, "Cooperation without communication: Multiagent schema-based robot navigation," *Journal of Robotic Systems,* vol. 9, pp. 351-364, 1992.

[83] D. Goldberg and M. J. Matarić, "Coordinating Mobile Robot Group Behavior Using a Model of Interaction Dynamics," in *Proceedings of the Third Annual Conference on Autonomous Agents*, New York, NY, USA, 1999.

[84] D. Rus, B. Donald and J. Jennings, "Moving furniture with teams of autonomous robots," in *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, 1995.

[85] M. J. Mataric, M. Nilsson and K. T. Simsarin, "Cooperative multi-robot box-pushing," in *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, 1995.

[86] J. M. Angel, F. D. la Rosa and G. Bravo, "Cooperative architecture for multi-agent systems in robotic soccer (CAMASS)," in *IX Latin American Robotics Symposium and IEEE Colombian Conference on Automatic Control, 2011 IEEE*, 2011.

[87] O. S. Keong and K. Omar, “A review of multi-agent systems in building multiple soccer-playing intelligent robots,” in *2011 International Conference on Pattern Analysis and Intelligence Robotics*, 2011.

[88] F. Ehlers, “Multi-agent Teams for Underwater Surveillance,” in *2010 International Conference on Emerging Security Technologies*, 2010.

[89] S. Zheng, X. Zhang and S. Sheng, “Simultaneous fault detection and control protocol design for coordination of multi-agent systems,” in *2017 29th Chinese Control And Decision Conference (CCDC)*, 2017.

[90] H. Su, X. Wang and Z. Lin, “Flocking of Multi-Agents With a Virtual Leader,” *IEEE Transactions on Automatic Control,* vol. 54, pp. 293-307, Feb 2009.

[91] O. Linda and M. Manic, “Fuzzy manual control of multi-robot system with built-in swarm behavior,” in *2009 2nd Conference on Human System Interactions*, 2009.

[92] C. Lei, W. Sun and J. T. W. Yeow, “A distributed output regulation problem for multi-agent linear systems with application to leader-follower robotś formation control,” in *2016 35th Chinese Control Conference (CCC)*, 2016.

[93] J. Contreras and F. F. Wu, “Coalition formation in transmission expansion planning,” *IEEE Transactions on Power Systems,* vol. 14, pp. 1144-1152, Aug 1999.

[94] N.-F. Xiao and S. Nahavandi, “Market organization model-based multi-agent robotic system,” in *The 2002 International Conference on Control and Automation, 2002. ICCA. Final Program and Book of Abstracts.*, 2002.

[95] I. Praca, C. Ramos, Z. Vale and M. Cordeiro, “MASCEM: a multiagent system that simulates competitive electricity markets,” *IEEE Intelligent Systems,* vol. 18, pp. 54-60, Nov 2003.

[96] V. Krishna and V. C. Ramesh, “Intelligent agents for negotiations in market games. I. Model,” *IEEE Transactions on Power Systems,* vol. 13, pp. 1103-1108, Aug 1998.

[97] V. S. Koritarov, "Real-world market representation with agents," *IEEE Power and Energy Magazine,* vol. 2, pp. 39-46, July 2004.

[98] H. Liu, H. Yu, Y. Li and Y. Sun, "A role modelling approach for crowd animation in a multi-agent cooperative system," in *Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2011.

[99] Y. W. Fu, J. H. Liang, Q. P. Liu and X. Q. Hu, "Crowd Simulation for Evacuation Behaviors Based on Multi-agent System and Cellular Automaton," in *2014 International Conference on Virtual Reality and Visualization*, 2014.

[100] M. Ntika, P. Kefalas and I. Stamatopoulou, "Multi-agent system simulation of nano-robotic drug delivery in tumours of body tissues," in *2013 17th International Conference on System Theory, Control and Computing (ICSTCC)*, 2013.

[101] "Foundation for Intelligent Physical Agents web page," [Online]. Available: http://www.fipa.org/.

[102] "Jade Multi-Agent System," [Online]. Available: http://jade.tilab.com/.

[103] B. Rafael H., J. F. Hübner and M. Wooldridge, Programming multi-agent systems in AgentSpeak using Jason, John Wiley & Sons, 2007.

[104] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW 96 Eindhoven, The Netherlands, January 22--25, 1996 Proceedings*, W. Van de Velde and J. W. Perram, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 42-55.

[105] A. Ricci, M. Viroli and A. Omicini, "CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS," in *Environments for Multi-Agent Systems III: Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers*, D. Weyns, H. V. D. Parunak and F. Michel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 67-86.

[106] A. Omicini, A. Ricci and M. Viroli, "Artifacts in the A&A meta-model for multi-agent systems," *Autonomous Agents and Multi-Agent Systems,* vol. 17, pp. 432-456, Dec 2008.

[107] D. Weyns, M. Schumacher, A. Ricci, M. Viroli and T. Holvoet, "Environments for Multiagent Systems," Report AgentLink Technical Forum Group, Ljubljana, 2005.

[108] D. Scaradozzi and et.al, "Innovative technology for studying growth areas of Posidonia oceanica," in *IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS)*, 2009.

[109] S. B. e. a. Williams, "Monitoring of benthic reference sites: using an autonomous underwater vehicle," *IEEE Robotics & Automation Magazine,* vol. 19, no. 1, pp. 73-84, 2012.

[110] P. e. a. Zapata-Ramírez, "Innovative study methods for the Mediterranean coralligenous habitats," *Advances in Oceanography and Limnology,* no. 4, pp. 102-119, 2013.

[111] D. Scaradozzi, S. Zingaretti, L. Panebianco, C. Altepe, S. M. Egi, M. Palma, U. Pantaleo, D. Ferraris and F. Micheli, "DocuScooter: A Novel Robotics Platform for Marine Citizen Science," in *International Society of Offshore and Polar Engineers*, San Francisco, 2017.

[112] L. Sorbi, Z. Zoppini, S. Zingaretti and D. Scaradozzi, ""Robotic Tools and Techniques for Improving Research in Underwater Delicate Environment," *Marine Technology Society Journal,* vol. 49, no. 5, pp. 6-17, 2015.

[113] D. Costa, G. Palmieri, M.-C. Palpacelli, L. Panebianco and D. Scaradozzi, "Design of a Bio-Inspired Autonomous Underwater Robot," *Journal of Intelligent and Robotic Systems: Theory and Applications,* pp. 1-12, 2017.

[114] D. Scaradozzi, G. Palmieri, D. Costa, S. Zingaretti, L. Panebianco, N. Ciuccoli, A. Pinelli and M. Callegari, "UNIVPM BRAVe: A Hybrid Propulsion Underwater Research Vehicle," *IJAT,* vol. 11, no. 3, pp. 404-414, 2017.

[115] M. Wollridge and N. Jennings, “intelligent agents: theory and practice,” *The Knowledge Engineering Review,* vol. 10, no. 2, pp. 115-152, 1995.

[116] G. Conte, D. Scaradozzi, D. Mannocchi, P. Raspa and L. Panebianco, “Development and Testing of Low-Cost ASV,” in *The 26th International Ocean and Polar Engineering Conference*, Rhodes, Greece, 2016.

[117] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler and A. Y. Ng, “ROS: an open-source Robot Operating System,” in *ICRA Workshop on Open Source Software*, 2009.

[118] G. Conte, D. Scaradozzi, D. Mannocchi, P. Raspa, L. Panebianco and L. Screpanti, “Development and Experimental Tests of a ROS Multi-agent Structure for Autonomous Surface Vehicles,” *Journal of Intelligent & Robotic Systems,* Oct 2017.

[119] L. Panebianco, S. Zingaretti, N. Ciuccoli, C. Altepe, S. M. Egi, F. Micheli and D. Scaradozzi, “Procedures and Technologies for 3D Reconstruction with Divers of Underwater Archaeological Sites and Marine Protected Areas,” in *Metrosea 2017*, Napoli, 2017.

[120] L. Sorbi, D. Scaradozzi and G. Conte, “Geoposition data aided 3D documentation of archaeological and biological sites,” *Proceedings of Meetings on Acoustics,* pp. 1-9, 2012.

[121] M. W. Maier, “Architecting principles for systems-of-systems,” *Systems Engineering,* vol. 1, pp. 267-284, 1998.

[122] D. Scaradozzi, L. Sorbi, S. Zingaretti, M. Biagiola and E. Omerdic, “Development and integration of a novel IP66 Force Feedback Joystick for offshore operations,” 2014.

[123] E. Alonso, N. Karcanias and A. G. Hessami, “Multi-Agent Systems: A new paradigm for Systems of Systems,” in *ICONS 2013 : The Eighth International Conference on Systems*, 2013.

[124] M. Colledanchise and P. Ögren, “How Behavior Trees modularize robustness and safety in hybrid systems,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.

[125] “JSON reference site,” [Online]. Available: https://www.json.org/.