

MeRMaID

Middleware for Multiple Robot Intelligent Decision-Making

M. Barbosa¹ P. Lima¹ J. Sequeira¹

¹ Institute for Systems and Robotics, Instituto Superior Técnico, 1049-001 Lisboa, Portugal

Abstract—In this paper we introduce MeRMaID (Multiple-Robot Middleware for Intelligent Decision-making), a robot programming framework whose goal is to provide a simplified and systematic high-level behavior programming environment for multi-robot teams. MeRMaID's 2 layers (support and functional architecture) are described in detail. The functional architecture, built over the support layer, constrains some of the developer options, so as to guide him/her towards building better and maintainable code. MeRMaID is generic enough so that the developer can implement/use several types of algorithms/methods throughout the various architecture components to accomplish a specific task, and even choose to implement his/her own functional architecture. Examples of applications to an European project on ubiquitous networked sensor and robots and to a RoboCup Middle Size League soccer team are presented in detail.

Index Terms—Middleware, Functional Architecture, Multirobot Systems, Service-Oriented Architecture

1 INTRODUCTION

Modern robotics systems have reached a high level of complexity, including centralized and decentralized implementations of multiple sub-systems, requiring mutual interactions.

Most of the software architectures currently used with robotic systems enable creating robot testbeds, reducing the user burden concerning communications and data sharing, but requiring the user to define the information flow, decision components and execution flow. On the other hand, several behavior coordination methods are available [1], but usually they are not associated to a programming environment where robotic tasks are described at a reasonable level of abstraction (e.g., programming behaviors as state machines where states represent primitive actions). The ever increasing level of autonomy demanded to robotic systems is naturally leading to an increase in the complexity of such sub-systems. The first

step towards structuring the implementation of complex robotic systems lead to the concept of architecture that emerged during the 80's, ranging from hierarchically organized structures of concepts to networks of basis functionalities. More recently, the robotics community started to explore synergies with software engineering (see for instance the recent [2]).

Well known paradigms, such as the subsumption architecture [3] promised an easy way to the creation of intelligent robots just by letting basic functions compete to influence the actuators of a robot. In its basic form, subsumption is of limited use but it was very important in establishing guidelines for the development of software for complex robotic systems. Basic building blocks, implementing primitive behaviors for the robotic system, are connected as nodes in a network with the information flowing among them being controlled by dedicated blocks. Conceptually, this is a two tier architecture with the interconnections and control blocks forming one layer and the primitive behaviors forming another.

Whether it is between levels of architecture or among primitive behaviors, generic interfacing between sub-systems is becoming increasingly important in Robotics. In fact, the similarity between architectures in all do-

• This work was supported by the European Project FP6-2005-IST-6-045062-URUS, and ISR/IST plurianual funding through the POS_Conhecimento Program that includes FEDER funds.

mains has been recognized by some authors [4] and it is a useful conjecture to claim that interface technologies are used by any architecture.

Generic interfacing can be seen as a component that integrates all the other components, with the responsibility of connecting and synchronizing them, translating and adapting information. Moreover, the increasing complexity of systems fosters the development of sophisticated interfacing components targeting (i) platform independence, (ii) enhanced scalability, (iii) development process simplification, (iv) real-time performance, (v) integration with existing infrastructure (vi) promoting software reuse, (vii) programming language independence. In Robotics parlance these components and strategies are nowadays known as *middleware*, following similar designations in computer engineering, referring to computer software that connects software components or applications.

The development of robotics middleware has followed that of architectures, often without a distinct identity. A huge amount of work has been produced in the last two decades on robotics architectures. For example, the DAMN architecture [5] is a collection of independent modules, implementing distributed behaviors, communicating with a centralized arbiter. In the CAMPOUT architecture [6] the building blocks include device drivers and communication layers in addition to primitive behaviors and behavior composition strategies. Satake *et al* [7] propose a networking concept for robots based on the LACOS context manager. The network has a standard node organization. Each node is composed of a robot and a LACOS interface to the network bus. The LACOS interface contains an application interface, a query issue engine, a result collector and a context database with a corresponding manager. The application interface takes a query from an application in the robot and takes it to the query engine. The query engine broadcasts all the LACOS interfaces in the network for contexts referred in the query. Such broadcasts are received and processed by the context data manager, put in the network, grabbed by destination query result collector, and then forwarded to the application that originated the request.

Among those referring explicitly middleware components, Woo *et al* [8] proposed a three tier architecture with an application layer, and infrastructure services layer and a middleware layer. The application layer contains the functional blocks related to single robot activities, e.g., path planning. The infrastructure layer provides the network services. The middleware layer handles the communication between services. Taira and Yamasaki [9] describe a five layer architecture in which the middleware concept is easily recognized. It encom-

passes device managing, control, networking, integrating and application layers. The device management layer interfaces the physical devices with the rest of the architecture and contains the low level control strategies. The control layer acquires high-level data, e.g., face and voice recognition and navigation data. The network layer hides the communication interfaces among the modules. The integrating layer contains a map manager, a global planner and a database system. The application layer is used to define the tasks for the overall system using abstract commands.

The middleware concept is tightly connected to communications and networking software technologies. In fact the term middleware has also been misleadingly used to refer to the low/middle level communications technologies, e.g., Microsoft .NET, Java RMI, SOAP, and even TCP/IP and UDP/IP, used by architectures, forgetting the integration component. Kuo and MacDonald [10] describe a CORBA based architecture with two layers: infrastructure and service. The infrastructure contains the classes that form the foundations of the service layer but can also be used by user applications. The service layer provides the high-level services the users can deploy in their applications. The three tier architecture in [8] uses a CORBA broker to provide the protocol between services and query facilities. CORBA was chosen over Microsoft's DCOM and Sun's Java/RMI, based on the availability, existing implementations and public domain licencing policies associated to CORBA products. Schorr *et al* [11] described an object oriented architecture to control a distributed surgical robotics system. This architecture also uses the CORBA framework to implement object distribution. The overall system includes multiple computing hosts over different operating systems and software languages. Making the communications among distributed services transparent to developers has also been the goal of the community developing YARP [12]. The YARP framework encompasses a library of functions for creation and management of network communications and communications server facilities. YARP allows the user to abstract from most of the details of the networking infrastructure and hence can handle the communications in the functions wrapping the robotics specific components.

Middleware frameworks for systems integration in Robotics are relatively few. The Miro framework [13] is a distributed object oriented framework for robot control also based on CORBA. Miro represents a middleware layer for autonomous robots, providing network transparency, event based publisher, logging facilities, and sensor and actuator services. Saphira/Aria has been used in the DARPA Software for Distributed Robotics [14]

addressing the distributed networking of large size (100 plus) robot teams operating in indoor missions. The Player/Stage project [15] proposes a relatively low level tool systems integration. The top programming level, called the Stage, allows the simulation of robots, sensors and environmental objects. The second level, Player, is built directly over TCP socket technology. Client libraries provide the wrapping interface to multiple common languages such as C, C++, Java and Common Lisp. The ART proposal [16], used in the RoboCup Middle Size League (MSL), is supported on ETHNOS, a dedicated operating system that acts as the middleware layer interconnecting the different robots in the team. A specific networking protocol, extending UDP, was used by ETHNOS. The extension of operating systems to robotics tools is a natural step as robots are just a locomotion extension of computers. Microsoft Robotics Studio (MSR) [17] is a further example of a distributed service oriented architecture, supported on the Windows operating system and the .NET framework. MSR focus on visual programming and embeds a simulation tool and copes with a variety of programming languages and robotic devices.

The aforementioned conjecture on the similarities between robotics architectures is currently leading to joint efforts on standardization. The Robot Standards (RoSta) project is an attempt to coordinate the development of standards and reference architectures for robotics. Its scope encompasses the definition of a domain glossary/ontology, specification of a reference architecture and middleware for mobile manipulation and service robots as well as the formulation of benchmarks for mobile manipulation and service robots [18].

In this paper we introduce MeRMaID (Multiple-Robot Middleware for Intelligent Decision-making), a robot programming framework whose goal is to provide a simplified and systematic high-level behavior programming environment for multi-robot teams, supported on a middleware support layer [19]. This middleware layer uses the Service concept as its basic building block. The functional architecture, built over the support layer, constrains some of the developer options, so as to guide him/her toward building better and maintainable code. In MeRMaID, a high-level functional architecture is presented to the developer, who must develop (or reuse) components that fit the functional architecture. This way, MeRMaID ensures that separately developed components will be more easily assembled together at integration time. Moreover, MeRMaID is generic enough so that the developer can implement/use several types of algorithms/methods throughout the various architecture components to accomplish a specific task. The use of

this architectural layer is not mandatory, therefore the developer may always build its application over the support layer, without any constraints on how to connect subsystems.

MeRMaID is an evolution of previous versions of the software architecture of the ISR/IST RoboCup Soccer MSL team ISocRob [21], which have been used since year 2000 with a team of 5 real cooperative robots. Meanwhile, MeRMaID::support was extended to an European research project, URUS (Ubiquitous Robots in Urban Scenarios) [22], where it supports the integration of several partner contributions concerning a camera and robot network, including vision, (cooperative) navigation, (cooperative) perception, path and task planning and execution.

The paper is organized as follows. Section 2 details the support layer of the MeRMaID middleware. Section 3 describes the functional architecture built over the support layer. Case studies on the European project URUS and the SocRob project (ISocRob RoboCup MSL team) are detailed in Section 4. Section 5 presents the conclusions and future developments.

2 MERMAID::SUPPORT

Developers normally expect middleware packages to offer building blocks for their code. MeRMaID adopts the Service-Oriented Architecture (SOA) architectural design pattern. Therefore the central concept of MeRMaID's building blocks is that of Service. A Service is, in its essence, a guardian of a specific functionality, i.e. a piece of code that is able to provide some kind of useful function to the system and that is willing to provide it to those who need it. Given this definition, it can be argued that almost everything can be seen as a single service, from a complex software solution that controls a full team of robots down to a very simple operation such as, for instance, the sum of two integers. This is a problem of partitioning that is addressed in Section 3 where a partitioning strategy for software controlling robots is proposed.

MeRMaID::support provides the core infrastructure on which MeRMaID::functional-architecture builds upon, by specifying and/or implementing a number of concepts, protocols and tools. In Figure 1 an overview of the structure of both levels of MeRMaID can be seen.

In this section we will describe the facilities offered by MeRMaID::support, namely the implementation of several concepts introduced in the "Reference Model for Service Oriented Architecture" developed by the OASIS group [23]. interaction mechanisms between Services, system description files as well as several other auxiliary

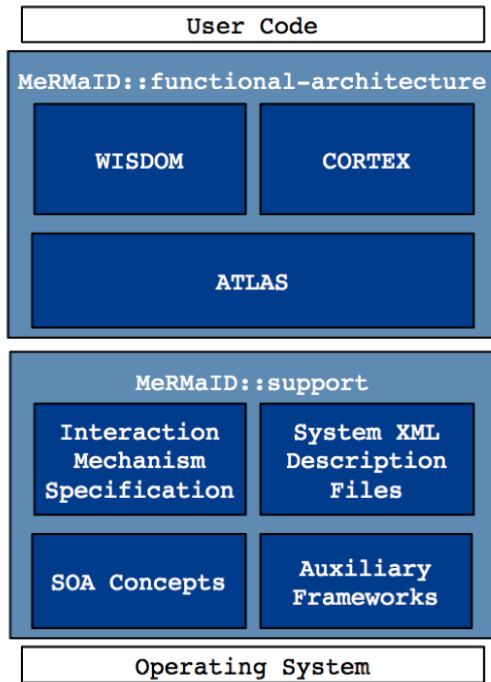


Fig. 1. General structure overview of MeRMaID::support and MeRMaID::functional-architecture.

functions provided by MeRMaID::support in the form of frameworks.

2.1 Service Interaction Mechanisms

A Service-Oriented Architecture relies on the ability of Services to interact with each other in order to have the system performing as expected. Therefore, it is necessary to define which interfaces Services offer and how should they be used. The range of issues to be dealt with concerning communication between Services, or any two software components in general, covers the range of all of the OSI model [24]. MeRMaID::support concentrates on the top layer, the "Application Layer" and partially on the "Presentation Layer". It uses YARP [12] as a communications library which deals with (de)serialization of basic data types and point-to-point communication which works over standard TCP/IP protocols.

YARP was chosen as the communication library with which all communication protocols were implemented as one of the goals of the design of the communication stack was to make it possible to interact with programs that are developed without using MeRMaID. YARP offers a very simple concept of "port" with which all communication is done. The protocol implemented on top of YARP is very simple enabling code developed without MeRMaID to be easily made compatible. This is of great importance

given that normally developers would like to reuse existing code without having to change their programs to cope with a new programming environment. Besides this issue, it is possible to implement MeRMaID's interaction mechanism with industry-standard technologies such as CORBA or Web Services.

MeRMaID::support defines two interaction mechanisms between Services:

- **service request:** a single-shot interaction between Services that consists in a request followed by a reply
- **data feed:** a continuous stream of information produced by a single Service and that may be read by any other interested Service.

2.1.1 Service Request

The Service Request interaction mechanism deals with single-shot interactions between two services: one that takes the initiative on starting the interaction, the Requester Service, and another which receives the request and should process it and send a reply, called the Target Service. This mechanism is suited for typical request/reply and polling interactions between software components.

2.1.2 Data Feed

The Data Feed interaction mechanism deals with continuous streams of information as well as information that the receiver does not know when it will be produced. The Service that produces data is called the Writer Service while the Services that receive it are called the Reader Services. For each Data Feed there is only one Writer Service and an indeterminate number of Reader Services (possibly even zero). Therefore it is well suited for distributing data from sources that produce it continuously as well as for event-based interactions (in which the receiver does not know when will the event occur).

2.2 System description files

One important characteristic of a Service Oriented Architecture is that there needs to be a way by which Services (and their developers) know which other Services exist and which functionality do they provide. This is done via description files. MeRMaID employs three types of description files:

- **Entity Description Files** describe which physical entities exist in the system and which Services are available in each of them, describing how the system is deployed.

- **Service Type Description Files** describe which types of Services exist, stating which interfaces are offered by each Service Type.
- **Data Type Description Files** describe which data types are used in the interaction between Services, stating how each data type is composed and what is the meaning of each individual field.

These description files are also very important in coordinating the work of several different teams of developers. Given that a team of developers wants to develop a functionality and make it available to the whole system, i.e. they want to develop a Service, they can completely define how it will interact with other Services by making their description files available. The other developers are able to use this Service by just reading the description files. These have information that allows MeRMaID::support to validate all interactions between Services as well as to do all the tedious work of communicating with other Services. The description files also have human-readable fields where extra information on the semantics behind the use of a Service may be stated.

2.3 Auxiliary Frameworks and Tools

MeRMaID::support offers some other frameworks and tools that are useful for the development of Services:

- **Memory Management Framework** Collection of smart pointer types that help remove the burden of memory management from the developer. Currently implemented smart pointers use reference counting and copy-on-write strategies.
- **IO Framework** Input/Output operations to hardware devices are supported with a programming model identical to that of Service interaction. The developer may read and write raw data to a device just like as if it was another Service in the system. All IO operations are handled asynchronously, without any need of a Service to wait for an IO operation to complete.
- **Logging Framework** A useful logging framework that collects information considered relevant by the Service developer. The logs may be output to a terminal, a file or even as a Data Feed to be read by other Services.
- **MeRMaID Loader Generator Tool (mlgen)** This tool generates code that is able to deploy a Service. Once the developer has written the code for the Service being developed, mlgen generates executable code that correctly deploys the Service based on a given configuration and following what was defined in the System Description Files.

- **MeRMaID Ping Tool (mping)** This tool allows to easily ping the available Services on the system. It reads the System Description Files and tries to ping all declared Services. A successful ping to a Service implies that it is running (i.e. not in a deadlock or zombie mode) and that its interaction mechanisms are working correctly. The ping tool contacts the Services using the Service Request interaction mechanism. MeRMaID::support provides a default implementation of the code needed to answer a ping request to every Service.

3 MERMAID::FUNCTIONAL-ARCHITECTURE

MeRMaID::support offers a generic framework with which any developer can develop Services and make them interact with each other. Although some requirements from the robotics domain were taken into account (like having interaction mechanisms support both polling and event-based interactions, as well as simplifying access to hardware devices) it is almost domain-independent. One important issue that is not addressed is the definition of which Services should exist and at what granularity. This is what is addressed by MeRMaID::functional-architecture.

In MeRMaID, we have defined an architecture for decision making in multirobot systems, that takes into account the main subsystems at the individual level, as well as the requisites for cooperation, concerning different aspects, such as cooperative perception and teamwork, providing means for different decision-making strategies (e.g., predicate-based, event-based, mixed).

In this section, we define the main concepts and components of MeRMaID functional architecture, and the developed robot programming frameworks that use them, with a special focus on the Petri net representation of (possibly cooperative) plans, also used to execute them.

3.1 MeRMaID Concepts

The most relevant concepts in MeRMaID are *roles*, *behaviors*, *primitive actions*, *navigation primitives*, *predicates*, and *events*. Their definitions follow:

- **Navigation primitive** is a guidance algorithm which, based on the current and target robot postures (position plus orientation) and current self-localization estimate, computes the required wheel speeds to move the robot from the current to the target position avoiding obstacles on the way.
- **Primitive action** is the atomic element of a behavior, which can not be further decomposed. It usually consists of some calculations (e.g., determination

of the desired posture) plus a call to a navigation primitive or the direct activation of an actuator. Desirably, it is designed as a STA (Sense-Think-Act) loop, i.e., a generalized view of the closed-loop control system concept. This means that our functional architecture assumes primitives that moves the robot towards its goal while avoiding obstacles, rather than having one primitive that moves towards the goal and another that avoids obstacles.

- **Behaviors** are defined as "macros" of primitive actions grouped together using some appropriate representation. For instance, a behavior may consist of a state machine which states represent primitive actions, and transitions between states have associated events, but it could also be defined by a fuzzy decision-making algorithm based on fuzzy rules, used to select sequences of primitive actions to be executed.
- **Predicates** are Boolean relations over the domain of world objects, e.g., $see(x)$, where x can be ball, pole, or field_line, in the soccer domain, or $near(r,x)$, where r is any of the team robots, and x can be any world object.
- **Event** is, in general, an instantaneous occurrence which denotes a state change (e.g., of a variable, of a robot). In MeRMaID, we limit the event definition to changes of (logical conditions over) predicates from True to False or False to True (and we call these *internal events*), though we include events received from sources external to a robot through a communication channel. These *external events* do in fact meet our global definition, as they could trigger a data change which would trigger a predicate, resulting in an event occurrence, but in practice we do not do it this way, in order to simplify the implementation. Examples of internal events in robot soccer are: event `lost_object` occurs when the predicate `has(object)` changes its value from True to False, and vice-versa for event `got_object`; event `found_object` occurs when the predicate `see(object)` changes from False to True. Example of external events in robot soccer are signals sent by the referee box telling the robots to stop, execute a goal, corner kick or a throw-in.
- **Roles** are subsets of behaviors, defined over the set of available behaviors. When a role is selected (e.g., Attacker, Defender, GoalKeeper in the soccer domain), a new set of behaviors becomes enabled for selection by the behavior coordination mechanism. In practice, a role constrains the possible options for a robot selection of behaviors, effectively constraining the overall behavior displayed by the robot.

Note that roles do not form a partition over the set of available behaviors, since there are behaviors that may be shared by more than one role (e.g., `GetClose2Ball` for the Attacker and Defender roles above).

MeRMaID functional architecture is divided in 3 major building blocks, each of them having several components:

- **ATLAS** (i.e., the subsystem that supports the whole system): is responsible for the tasks most directly related to the robot's environment: sensing and acting.
 - Devices: handles the low level interface with physical-world devices, both actuators and transducers (e.g. motors, sonars, cameras).
 - Sensors: obtain information from the transducer devices (e.g. odometry, obstacle location, ball position). Note that, in MeRMaID's conceptualization, one transducer may correspond to several sensors: a typical example is the vision camera transducer and the sensors one can develop by processing one or several acquired images from it, e.g., to recognize specific objects, to find straight lines, to discriminate colors
 - Information Fusion: fuses information from several sensors (which can be sensors onboard the robot or from external sources)
 - Primitive Actions: see definition in the previous section
 - Navigation Control: refers to navigation primitives defined in the previous section
- **WISDOM** (i.e. a very relevant requirement for intelligence to be displayed): acts as a central point of information storage in the format of raw, processed and symbolic data
 - World Info: store general purpose high-level data, relevant for predicate evaluation. World Info may hold information originating from other robots.
 - Predicate Manager: handles predicate Boolean values, based on the current raw and processed data values.
- **CORTEX** (from CoORDinator, TEAm organizer, eXecutor): the decision making module
 - Team Organizer: responsible for the actual organization of the team in terms of roles. It activates roles in each of the team robots
 - Behavior Coordinator: responsible for behavior selection and coordination. It activates a behavior from the set of behaviors available for the

currently selected role

- Behavior Executor: responsible for behavior execution. It activates Primitive Actions, for the currently selected behavior.

MeRMaID main components, i.e., Devices, Sensors, Sensor Fusion, Navigation Control, Primitive Actions, World Info, Predicate Manager, Team Organizer, Behavior Coordinator and Behavior Executor, are implemented as Services running in the MeRMaID::support layer. Figure 2 shows MeRMaID's reference functional architecture, by establishing a diagram of component (i.e., Services) connections that constrains the flow of information among them. Raw information about the surrounding environment of the robots is acquired by transducer Devices, processed by sensors and possibly fused among several sensors of the same robot or sensors from different robots in the Sensor Fusion component (see [25] for an example on fusing information from different robot cameras on a moving object position and velocity). The relevant fused information is stored in the World Info, either as numeric or symbolic data (examples in soccer robots are the estimated ball position, the estimated robot posture, or proposition $\text{see}(\text{ball})$). The Predicate Manager is in charge of, upon notification by the World Info of changes on data relevant for the computation of the logical value of a predicate, updating the predicate value. This information, as well as changes in the predicate boolean values (i.e., events) can be accessed by the CORTEX components, so as to take decisions on selection among roles (Team Organizer), behaviors (Behavior Coordinator) and primitive actions (Behavior Executor). Primitive actions often require Navigation Control to compute the wheel speed set points that will guide the robots to some desired location (including dynamically changing target locations, e.g., while tracking a ball). These are sent to the actuator Devices, and the loop is closed through the environment.

Cooperation among team robots can occur in different forms, and typically requires communication among robots. Communication can be either implicit or explicit. In the former case, one robot uses its sensors to perceive its teammate(s) behavior and act coordinately with them. In the (most usual) second case, wireless communications are used. The later requires communications management, explained in Section 2. We will explain here how does MeRMaID handle cooperative issues in two important cases: cooperative perception and cooperative task execution.

Cooperative perception is handled by the Sensor Fusion and World Info components. The information gathered on some specific item (e.g., an object position) by

one or more teammates goes to their World Info, which, in the explicit communication case, send that information to the World Info of another teammate requiring it (e.g., because it does not see the object or wants to obtain a more accurate estimate of its position). Then, the Sensor Fusion component at this teammate fuses the World Info information from its teammates with the information acquired by its own sensors. This may include, e.g., procedures to check agreement among sensors, when to fuse disparate estimates.

Cooperative task execution depends on the exchange of synchronization and commitment signals among robots [26]. Such signals can lead to changes in predicate logical values at the World Info components of the robots involved in the teamwork, which can then be used by the Team Organizer and Behavior Coordinator to take decisions on role and behavior selection, respectively, taking into account the current state of their teammates.

3.2 Petri net programming framework

The description of MeRMaID's functional architecture is intended to be a guide for the system developer, defining which components should exist and how should they interact with each other. The developer is still free to choose the concrete implementation of each component. Currently, we have developed 3 different possible implementations for components inside CORTEX: Finite State Machines, Petri Nets and Fuzzy Logic-based Behavior Arbitration[27]. Here we will focus on up-to-date MeRMaID's most widely used framework for CORTEX: Petri nets. While describing how do Petri nets implement the CORTEX, we will also clarify some of the concepts in the previous subsection, by materializing them with concrete examples.

Petri nets represent a class of Discrete Event Systems (DES). These are systems with a discrete set of states, whose dynamics is driven by the occurrence of events which cause transitions between states. The events occur asynchronously and typically DES do not model the event generation mechanism. Petri nets are defined by a 5-tuple (P, T, A, M_0, W) , where:

- P is a set of places
- T is a set of transitions
- A is a set of arcs, connecting transitions to places and places to transitions.
- $M_0 : P \rightarrow N$ is an initial marking, assigning to each place $p \in P$, $n \in N$ tokens.
- $W : A \rightarrow N^+$, known as the set of arc weights, assigns to each arc $a \in A$ some $n \in N^+$, denoting how many tokens are consumed from a place by a transition, or alternatively, how many tokens are produced by a transition and put into each place.

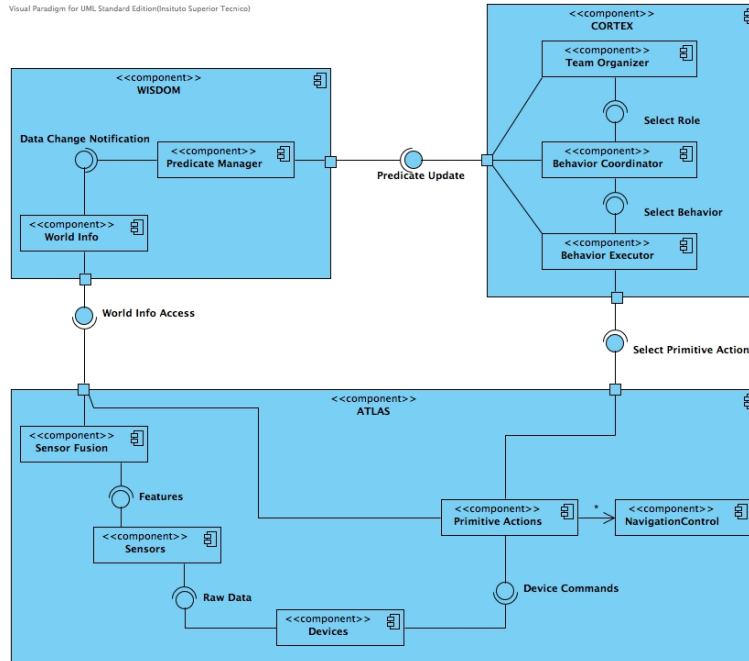


Fig. 2. MeRMaID functional architecture component diagram.

The Petri nets used to model behaviors have all arc weights equaling 1, therefore the weights are not represented in the Petri net graphs. In this work, behaviors are modeled by Petri nets as follows:

- each place in the Petri net is labeled by an associated primitive action or by a predicate;
- each transition in the Petri net is labeled by an event, as defined in the previous subsection.

A token in a place means that the primitive action associated to that place is currently active (i.e., it is running) or that the predicate labeling that place is **True**. Transitions are enabled when all its input places have at least one token each, meaning that the pre-conditions for the next step are satisfied. A transition is fired if its enabled and the associated event (if any) occurs.

Though the description above refers to Petri nets representing behaviors, it could refer to Petri nets representing roles, if places are now labeled (besides predicates) by behaviors, instead of primitive actions. Therefore, one can build a Petri net hierarchy to model the whole CORTEX in the previous subsection. Examples of soccer robots CORTEX modeled using Petri nets are depicted in subsection 4.2. By using both predicates and events, the decisions on role, behavior and primitive action selection can be either event-based and/or predicate-based (logic-based) and/or on a mix of the two. This is one of the reasons for selecting Petri nets as our default and favorite framework. Other reasons are:

- Petri net languages (languages marked by Petri

nets) are a superset of regular languages (languages marked by Finite State Automata), mainly due to Petri nets memory and concurrency distinctive features. Therefore, the set of possibly modeled roles and behaviors is potentially richer when using Petri nets;

- Petri nets enable distributed state modeling, i.e., one can start with simple models (e.g., a primitive action and its pre-conditions) and build more complex ones (e.g., a behavior Petri net out of several primitive action Petri nets);
- several tools for Petri net formal verification are available. Formal verification is useful for programming, but stochastic performance evaluation can also be achieved supported on widely available bodies of theory, methods and tools for Stochastic Petri nets: a stochastic model of primitive actions, their times between failures, and success probabilities can be built readily from the non-stochastic model and analyzed beforehand, even using closed form Markov Chain analysis methods, under some conditions.

4 APPLICATIONS

4.1 Case study: URUS project

The European URUS project, started December 2007, aims at deploying a Networked Robot System (NRS) in real urban scenarios in the city of Barcelona, Spain. The system is composed by a set of cameras, connected

through internet, and multiple heterogeneous robots with onboard computational power, odometry, sonars, GPS, and laser range finder sensors [22]. Experiments on urban surveillance, and transportation and guidance of people and goods are scheduled within the project. The aim is to demonstrate that networked robots can interact naturally with people in urban environments, going beyond usability aspects traditionally considered in human-computer interaction.

The Institute for Systems and Robotics (ISR) at Instituto Superior Técnico (IST), Lisbon, developed a testbed for NRS, the *Intelligent Sensor and Robot Network* (ISROBOTNET), that enables testing a wide range of techniques related to perception, robot navigation, e.g., cooperative localization and navigation, cooperative environment perception, cooperative map building, as well as human robot-interaction, distributed decision making, and task and resource allocation. The ISROBOTNET is built over MeRMaID and enables fast integration of the heterogeneous subsystems involved in a NRS, in a fairly controlled environment. This way, later stages of testing in real urban scenarios benefit of the fact that integration is already fully functional, and the focus can then be turned to the decision-making, task allocation, human-robot interaction, perception and navigation subsystems.

Currently, the ISROBOTNET testbed is composed of an indoor area of around 160 m^2 with 10 webcams disposed around the ceiling such that some of the fields of view do not overlap. The cameras are distributed in 4 groups, each of which is managed by its own computer, namely for image acquisition. The managing computers are connected to the ISR/IST network and can be accessed by duly authorized external parties. Ongoing work will extend the number of cameras and the usable indoor space to include multiple floors. Robots will use the same elevators as ordinary people to move between floors. Besides the camera sensors, four Pioneer AT and one ATRV-Jr robots are available. Each of the robots is equipped with sonars, onboard cameras, laser range finder and is Wi-Fi connected to the network. Figure 3 shows a view of one of the floors at ISR/IST where the testbed is currently implemented and a map with the cameras field-of-view.

The URUS project includes 11 partners, from different institutions (universities, research centers and companies) and countries, each working on different software components. One of the URUS experiments that is featured to illustrate the use of NRS in urban settings considers a set of robots that has to act like stewards in a large urban area, closing the access to people to specific areas and helping them to move out. This is a typical task in multiple urban settings, e.g., when a zoo

is about to close for the day it is common that officers come to warn people that they should leave, and when it is necessary to temporarily close the accesses to some public area.

These characteristics require a special attention on the integration approach. MeRMaID::support is being used to integrate all the software developed and testing it at ISROBOTNET before deployment in Barcelona. The overall system is flexible enough to even integrate components developed away from the MeRMaID::support layer and using only a basis Interaction Mechanisms (specified in 2.1).

Among the advantages already clearly identified in using the MeRMaID framework are (i) a very low time to have third part components fully integrated in to the system (in the range of 1 hour average), and (ii) the automatic validation of the interactions among services, assuring the developer that all protocols and specifications are correctly followed.

The integration of functionality provided by image processing software connected to the camera network in this setup is an example of quick integration with software not developed using MeRMaID. The software to detect people and robots was developed by an independent team without any constraint on how they would later integrate it with the rest of the system at ISROBOTNET. Given that the software was able to detect events visible by the camera network that occur asynchronously, the Data Feed mechanism was chosen as appropriate to make this information available. In fact, the whole camera network can be seen as just a producer of a stream of asynchronous events. It was quite simple to insert a valid data feed mechanism in the camera network code. Developers only needed to write data to a YARP Port (which is YARP's fundamental building block for communications) and populate the system's description file with a declaration of what data they were exporting. This approach can be applied for any data producing software component in the system and it makes any needed type of data available to the whole system. Services developed using MeRMaID::support also benefit from simplified access and full (syntactic) validation of the sent data.

One of the first robotic tasks implemented at ISROBOTNET was to send a robot to meet a detected person. A simple Service for controlling the robot's motions was developed and it works in one of two states:

- **stopped** the robot is stopped, either because it was told to stop or it reached its goal position
- **moving** the robot is moving towards its goal position. When it arrives at this position its state will change to "stopped".

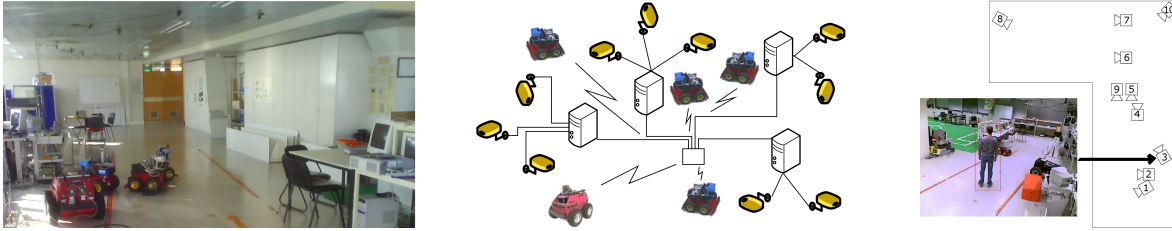


Fig. 3. Left: Partial view of the ISROBOTNET testbed, showing most of the cameras in the ceiling and the mobile robots equipped with several sensors. Center: Diagram of the connectivity links between cameras and robots, which have to share the limited network bandwidth. Right: The map of the environment showing the cameras' field-of-view and a picture taken from camera 3.

The robot's status is published periodically as a data feed, so that any component in the system to which this information might be important may receive it and behave accordingly. On the other hand, control of the robot is done using the service request interaction mechanism. This mechanism allows for confirmation that the sent request has been correctly received and processed. In the robot control service it is clear that the Service has its own internal dynamics which can be controlled, to some degree, by the interfaces it provides to the system.

Some additional Services were also developed using MeRMaID::support to provide task allocation and visualization capabilities. Although MeRMaID::functional-architecture is not being used in the URUS project nor in the current software deployment of the ISROBOTNET, MeRMaID::support provides the necessary tools to build the system, being the problem of partitioning it into Services the responsibility of the developer.

4.2 Case study: SocRob project

ISR/IST has been running an internal research project on cooperative robotics since 1997, denominated as Soccer Robots / Society of Robots (SocRob). The ISocRob team represents the experimental side of the project, and has been participating in RoboCup Middle-Size League since 1998. Over the years, the software architecture of the 5 robots soccer team has evolved into what is currently MeRMaID. In this subsection we will mainly focus on ISocRob's MeRMaID::functional-architecture, as a good illustration of MeRMaID's concepts at this layer.

The components at the 3 CORTEX layers are obviously motivated by the particular application:

- the Team Organizer's basic tactic selects among individual roles Attacker, Defender, Supporter and Goalkeeper, plus 2 additional cooperative behaviors, FoulTaker and FoulReceiver;
- the Behavior Coordinator selects among behaviors from the set of those enabled by the cur-

rently selected role: examples are MoveAwayFromBall, Move2StartPosition, OpponentFoulAttack, OurFoulDefender, ShortPassReceiver, LongPassTaker (there are currently around 20 behaviors running);

- the Behavior Executer executes the active behavior, by selecting its Primitive Actions as conditions change. Examples of Primitive Actions are ClearBall, CutDowntheAngle, InterceptBall, CatchBall, ApproachBall, Dribble (from around 80 active cases).

There are more than 100 predicates currently defined and used, e.g., Near(ball), ClosestTo(ball), IsLocalized(robot), Has(ball). Predicates are typically based on sensor data. As Sensors and Sensor Fusion components refresh the World Info, the Predicate Manager gets notified of data changes and updates the predicates registered in the variables that just suffered changes. As an example, if a robot vision system determines that the distance to the ball is larger than a given pre-defined threshold, predicate Near(ball) will be False. It will become True as soon as the distance between robot and ball shrinks below that threshold.

Some predicates require sharing information among robots, e.g., ClosestTo(ball) uses information from all team robots concerning their distance to the ball, and only one robot will make the predicate True, even if there is inconsistency among robots regarding distance measurements (e.g., due to a miscalibrated camera). This way, only one robot will move to the ball at a time.

Time tags are associated to Predicates, identifying the last time each predicate was changed. If the time lag between the latest update and the current time is larger than a threshold, the data is considered as unavailable, e.g., the robot does not "see" the ball if the predicate See(ball) has not been updated for a time longer than the threshold. Similarly, robot clocks are synchronized, and a robot will not consider predicate values sent by a teammate if the time tag differences among its latest

value and the teammate latest value of the predicate are over a given threshold.

Events result of changes in the Boolean value of a Predicate or from external signals. In the robot soccer domain, examples of the former are:

- Event `lost_ball` occurs when Predicate `Has(ball)` changes its value from `True` to `False`
- Event `got_ball` occurs when Predicate `Has(ball)` changes its value from `False` to `True`
- Event `found_ball` occurs when Predicate `See(ball)` changes from `False` to `True`.

Example of Events triggered by external signals are all the referee box events, e.g., `start_game`, `stop_game`, `throw_in`, `corner`.

CORTEX Petri nets are organized hierarchically. When the Team Organizer Petri net selects a Role, a Behavior Coordinator Petri net becomes active and starts selecting among available Behaviors, based on Predicate values (represented by predicate places — the Predicate is `True` when there is a token in its corresponding place) and events associated to transitions (if any). Whenever a Behavior becomes selected, its Behavior Executor Petri net similarly handles the Primitive Actions. This is illustrated in Figure 4. In this simplified Petri net example, the Team Organizer starts by selecting the `Supporter` Role, when Predicate `ShouldSupport` becomes `True`. The Petri net for the role is then selected and it starts with Behavior `BaseSupport` active. The Petri net switches to Behavior `Stop` if the referee box signals to stop the game. Then, it waits for a message from the referee box signaling a foul, and it switches to Behavior `FoullDefend` until the foul is taken (as signaled by the occurrence of an event). Once this happens, the Petri net returns to the initial `BaseSupport` Behavior. It may also get there from the `Stop` Behavior if the referee box signals a game (re)start. Behavior `BaseSupport` Petri net executor is depicted on the right of the figure: the `Supporter` either follows the ball at a given distance or searches for it, depending on whether it sees or does not see the ball, respectively. Associated primitive actions and predicates should be clear from this description and form the figure.

A good example of cooperative Behavior occurs when a foul is called. In case the foul is for our team, one of the team robots (the one with the Predicate `ClosestTo(ball)` `True`) will move to the ball to take the foul (it gets the `FoulTaker` role), while another robot (the one with the Predicate `SecondClosestTo(ball)` `True`) will move to some free spot on the field, near the ball, to receive it after the the `Taker` kicks it (he gets the `FoulReceiver` role). Both robots exchange synchronization (e.g., "I have reached the ball", "I am ready to receive the ball")

and commitment ("I am committed with the foul taking, will let you know if I will have to withdrawn") Predicates through message communication between their World Info components.

5 CONCLUSIONS AND FUTURE WORK

We have introduced MeRMaID and its 2 layers (support and functional architecture, as shown in Figure 1). MeRMaID::support is based on a service-oriented architecture, offering tools and functionality that ease the development of services and interactions among them, and corresponds to the only layer usually addressed by middleware for robots solutions. MeRMaID::functional-architecture, built over the support layer, provides a modular architecture where specific services and their interconnections are defined a priori, so as to guide the developer in building *maintainable* code (where modules correspond to specific subsystems and can be corrected and replaced, while keeping the same basic functionalities, and without affecting the other subsystems) and robotics-oriented software (as the functional blocks of the architecture were designed to handle multi-robot systems). The user is free not to use the pre-defined functional architecture and either to build up his/her own architecture on the top of the support layer, or simply work at the service level, without constraints of any kind. Nevertheless, MeRMaID's functional architecture is a step forward towards *interoperability* and *reusability*, since the clear definition of concepts such as Predicates, Events, Primitive Actions, Behaviors and Roles enables exchange of information among robots at a high abstraction level, and the ability to use these components in other applications and operating systems. In fact, the paper presents examples of MeRMaID applications to an European project on ubiquitous networked sensor and robots (URUS), involving many different European partners and expertise in robotic subsystems, and to a RoboCup Middle Size League soccer team (ISocRob), showing its versatility.

One further specific feature of MeRMaID is the availability of frameworks for task representation and execution, which represent particular instantiations of the CORTEX building block of the architecture. In this paper, we have focused in one such framework, Petri nets. Petri net representation of Roles, Behaviors and Primitive Actions provides a mathematically well-defined approach to task representation, which enables formal verification and performance analysis of plans, and a tool for actual task execution as well. Petri nets also lead to a natural approach to plan construction by composing simpler actions and behaviors, including cooperative plans, where several robots are involved.

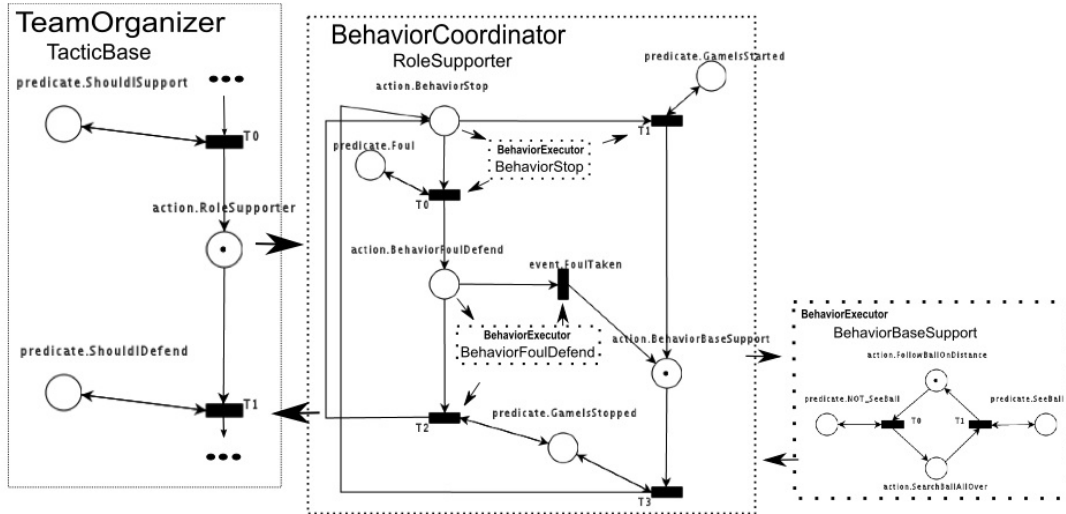


Fig. 4. Simplified hierarchical Petri net for ISocRob's CORTEX.

Future work will focus on further refinement of MeRMaID's 2 layers, namely in what concerns dependability. At the MeRMaID::support level, automatic service discovery and data distribution based on semantic information will be added as well as harmonization with standardization efforts such as the RoSta project. MeRMaID::functional-architecture will be applied to more applications and robotic sub-domains in order to further assess its suitability to successfully help developers build different types of software solutions for robotics.

REFERENCES

- [1] P. Pirjanian, "Behavior coordination mechanisms - state-of-the-art," Institute for Robotics and Intelligent Systems, University of Southern California, Los Angeles, CA, USA, Technical Report IRIS-99-375, 1999. 1
- [2] "IEEE Robotics & Automation Magazine," March 2009. 1
- [3] R. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, 1986. 1
- [4] A. Meystel and J. Albus, *Intelligent Systems: Architecture, Design, and Control*. Wiley Interscience, 2002. 1
- [5] J. Rosenblatt, "DAMN: A Distributed Architecture for Mobile Navigation," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 339-360, 1997. 1
- [6] H. Aghazarian, P. Pirjanian, P. Schenker, and T. Huntsberger, "An Architecture for Controlling Multiple Robots," webpaper accessed June 2006. [Online]. Available: www.nasatech.com/Briefs/Oct04/NPO30345.html 1
- [7] S. Satake, H. Kawashima, and M. Imai, "LACOS: Context Management System For a Sensor-Robot Network," in *Procs. of the 10th IASTED Int. Conf. on Robotics and Applications (RA 2004)*, 2004, pp. 160-165. 1
- [8] E. Woo, B. A. MacDonald, and F. Trépanier, "Distributed Mobile Robot Application Infrastructure," in *Procs. of the Int. Conf. on Intelligent Robots and Systems (IROS 2003)*, October 2003, pp. 1475-1480, las Vegas. 1
- [9] T. Taira and N. Yamasaki, "Functionally Distributed Control Architecture for Robot Systems," *Journal of Robotics and Mechatronics*, vol. 16, no. 2, pp. 217-224, 2004. 1
- [10] Y. hsin Kuo and B. A. MacDonald, "Designing a Distributed Real-time Software Framework for Robotics," in *Procs. of the Australasian Conference on Robotics and Automation*, December 2004, Canberra, Australia. 1
- [11] O. Schorr, N. Hata, A. Bzostek, R. Kumar, C. B. and Russell H. Taylor, and R. Kikinis, "Distributed Modular Computer-Integrated Surgical Robotic Systems: Architecture for Intelligent Object Distribution," in *Procs. of the Third Int. Conf. on Medical Image Computing and Computer-Assisted Intervention*, 2000, pp. 979-987. 1
- [12] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal on Advanced Robotics Systems, Special Issue on Software Development and Integration in Robotics*, vol. 3, no. 1, pp. 43-48, March 2006. 1, 2.1
- [13] H. Utz, S. Sablatnog, Enderle, and G. Kraetzschmar, "Miro - middleware for mobile robot applications," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 493-497, August 2002. 1
- [14] K. Konoldge, "DARPA Software for Distributed Robotics, Centibots Large Scale Robot Teams," SRI International, Tech. Rep. 2002-12-01, 2002. 1
- [15] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," in *Procs. of the Int. Conf. on Advanced Robotics (ICAR 2003)*, 2003, pp. 317-323, Coimbra, Portugal, June 30 - July 3. 1
- [16] D. N. L. Iocchi, M. Piaggio, and A. Sgorbissa, "Distributed Coordination in Heterogeneous Multi-Robot Systems," *Journal of Autonomous Robots*, vol. 15, no. 2, pp. 155-168, 2003. 1
- [17] "Microsoft robotics studio," Website accessed 2009. [Online]. Available: <http://msdn.microsoft.com/en-us/robotics/default.aspx> 1
- [18] [Online]. Available: <http://www.robot-standards.eu/> 1
- [19] M. Barbosa, N. Ramos, and P. Lima, "Mermaid - Multiple-Robot Middleware for Intelligent Decision-Making," in *Procs. of the 6th IFAC Symposium on Intelligent Autonomous Vehicles (IAV2007)*, 2007, toulouse, France. 1
- [20] C. R. Baker and J. M. Dolan, "Smart Streets for Boss - Behavioral Subsystem Engineering for the Urban Challenge,"

IEEE Robotics & Automation Magazine, vol. 16, no. 1, March 2009.

- [21] P. Lima, J. Santos, J. Estilita, M. Barbosa, A. Ahmad, and J. Carreira, "ISocRob 2009 team description paper," *Proceedings of RoboCup 2009 Symposium*, 2009. 1
- [22] A. Sanfeliu and J. Andrade-Cetto, "Ubiquitous networking robotics in urban settings," in *Workshop on Network Robot Systems. Toward Intelligent Robotic Systems Integrated with Environments. Procs. of 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2006)*, October 2006. 1, 4.1
- [23] OASIS, "Reference Model for Service Oriented Architecture 1.0," OASIS, Tech. Rep., August 2006. 2
- [24] H. Zimmermann, "OSI reference model—The ISO model of architecture for open systems interconnection," *Communications, IEEE Transactions on [legacy, pre-1988]*, vol. 28, no. 4, pp. 425–432, 1980. 2.1
- [25] J. Santos and P. U. Lima, "Multi-Robot Cooperative Object Localization - Decentralized Bayesian Approach," in *Proc. of RoboCup2009 Symposium*, July 2009, graz, Austria. 3.1
- [26] V. Ziparo, L. Iocchi, P. Palamara, D. Nardi, and H. Costelha, "Teamwork Design Based on Petri Net Plans," in *Proc. of RoboCup2008 Symposium*, July 2008, suzhou, China. 3.1
- [27] N. Ramos, P. U. Lima, and J. M. Sousa, "Robot Behavior Coordination Based on Fuzzy Decision-Making," in *Proc. of ROBOTICA2006, - 6th Conference on Mobile Robots and Competitions*, April 2006, guimaraes, Portugal. 3.2



Pedro U. Lima Pedro Lima got his Ph.D. (1994) in Electrical Engineering at the Rensselaer Polytechnic Institute, Troy, NY, USA. Currently, he is an Associate Professor at Instituto Superior Técnico, Lisbon Technical University. He is also a member of the Institute for Systems and Robotics, a Portuguese private research institution, where he is coordinator of the Intelligent

Systems group and a member of the Scientific Board.

Pedro Lima is a Trustee of the RoboCup Federation, and was the General Chair of RoboCup2004, held in Lisbon. He currently serves as President of the Portuguese Society of Robotics, for the 2009/10 period. He is also a founding member of the Portuguese Society of Robotics (2006), of the IEEE RAS Portugal Chapter (2005), and a senior member of the IEEE. He is the co-author of two books, regularly serves as member of international conferences program committees, and has coordinated national and international (ESA, EU) R&D projects. His scientific interests are in the areas of hybrid systems, discrete event systems and decision-making under uncertainty, mainly in their applications to complex large-scale systems, such as multi-robot systems. He has also been very active in the promotion of Science and Technology to the society, through the organization of Robotics events in Portugal, including the Portuguese Robotics Open since 2001.

João Silva Sequeira received the Graduation (5 years) and M.Sc degrees in 1987 and 1991, respectively, and the PhD degree in Electrical and Computer Engineering in 1999 from Instituto Superior Técnico (IST). He is currently an assistant professor at IST and a researcher at the Institute for Systems and Robotics (ISR), Lisbon, Portugal. He participated in a number of national and international R&D projects, published several papers in international conferences and journals and holds a patent on special purpose robots. His main research interests include robot architectures, hybrid systems, cooperative robotics, and human-robot interaction.



Marco Barbosa received his M.Sc. degrees in Information Systems and Computer Engineering from Instituto Superior Técnico (IST) in 2007. Currently, he is a research grantee at the Institute for Systems and Robotics (ISR), Lisbon, Portugal.

Marco Barbosa is a founding member of the Portuguese Robotics Society (2006) and has participated regularly in robotic

competitions such as the Portuguese Robotics Festival and the RoboCup tournaments. Currently he is also creating his own startup company which focuses on service robots and software development for robotic applications.

