MERMAID – MULTIPLE-ROBOT MIDDLEWARE FOR INTELLIGENT DECISION-MAKING

Marco Barbosa, **Nelson Ramos** and **Pedro Lima**

Instituto de Sistemas e Robótica
Instituto Superior Técnico
Av. Rovisco Pais, 1 – 1049-001 Lisboa, PORTUGAL

AbstractThis paper describes the basic concepts and features of 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 constrains, on purpose, some of the programmer's options, and can accept plans described by state machines, Petri nets and other types of decision-making algorithms, including fuzzy-logic decision-making and rule-based systems. Its current version enables concurrent multi-platform programming, modularity (for flexible module replacement and easy module edition/modification), and independence from robot hardware (since it includes an Hardware Abstraction Layer).

1. INTRODUCTION

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 (Pirjanian, 1999), 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).

Several robotic-oriented middleware solutions exist, but they tend to offer very low level functionalities to the end developer (e.g., YARP (Metta *et al.*, 2006) that is mainly focused on communication issues) or, even if they provide higher-level components, fail providing the developer with a precise guideline of how the system should be built and the recommended software and system architecture, e.g., MIRO (Utz *et al.*, 2002) and OROCOS (Bruyninckx, 2001).

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, which simultaneously constrains some of the developer options, so as to guide him/her toward building better and maintainable code. In MeRMaID, a high-level software architecture is presented to the developer, who must develop (or reuse) components that fit the 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.

MeRMaID is an evolution of previous versions of the software architecture of the ISR/IST

RoboCup Soccer Middle Size League team ISocRob (Lima *et al.*, 2003), which have been used since year 2000 with a team of 4 real cooperative robots. The current goal is to use this framework in wider application domains.

The paper is organized as follows: in Section 2 we list the most important requisites of robotic system middleware, which guided us in the development of MeRMaID, whose components are detailed in Section 3. Section 4 provides an application example, where the different components are involved. The paper ends with the major conclusions and a list of ongoing work, in Section 5.

2. SOFTWARE REQUISITES

In developing robotic systems, a considerable amount of the time spent is centered on software development. Typically, a robot is equipped with a standard computer running some Operating System (OS). While OSs already provide some kind of abstraction from the underlying hardware, this is not the ideal level in which roboticists would like to work on. Ideally, one would like to define the robot global behavior using some kind of high-level formalism. For instance, if the robot behavior is coordinated by a finite-state machine-based schema, the developer should only have to define the finite-state machine to control the robot. This would require all the software between the OS and the finite-state machine representation to be standardized, so as to be used independently of the underlying system architecture (comprising both software and hardware components).

Unfortunately, Robotics is still far from achieving this. Currently, most robot software is custom-made for each robotic system and for a particular application. In order to reach the goal of programming high-level behavior coordination methods and the corresponding behaviors, it would be useful to provide some standardization and formal organization of the software sitting in-between. Several projects attempted to create developer tools in order to accomplish this. These software components can be generally described as middleware for robotics application (in the sense that they stand in the middle of the OS and the representation of the behavior coordination formalism). The currently available middleware usually provides the developer with tools to address common useful functional requirements of the software components in robotic applications. These functional requirements are:

- Parallelism: more than one software component should be running simultaneously.
- Inter-component independency: components should be as independent from each other as possible. It should be possible to build and use a component regardless of which other components exist.
- Inter-component data sharing: components should be able to share data among themselves.
- Inter-component servicing: components should be able to request services from other components, in the sense that components may ask other components to do some kind of work for them.
- Platform independence: components should not need to know in which underlying platform they are running on. Usually the OS isolates software from the hardware, but components should also be isolated from the OS in order to be able to run the same component in several platforms.
- Localization independence: components should not need to know where a specific component that it uses is located and running.
- Modularity: components should be built and organized in a way to allow different implementations and algorithms to be used without affecting other components. For example, the developer should be able to replace the current navigation algorithm by a different one, without needing to make any change on any other component besides the one responsible for navigation of the robots.
- Multi-paradigm interaction: the middleware should not put constraints on how components should interact with each other (e.g. event-driven programming vs flow-driven programming)
- Real-time execution: certain robot tasks require to have real-time execution (e.g. components that detect that a robot is malfunctioning or if it is putting humans in danger).

Even if one has available a middleware that meets all of these requisites, there is no guarantee that a sound solution will be built. Decisions on how to organize the existing components are required. Therefore, it is desirable and useful to provide the developer with a supporting architecture that has already been tested and proven to work. Such architectures can be seen as software design patterns and their goal is to guide the developer in structuring his solution. Even though the desired middleware must be very generic in order to cope with all the functional requisites, it is possible (and desirable) to devise an architecture to handle items that all robotic systems share, such as sensors, actuators and control software. This is the main purpose of MeRMaID, and what distinguishes it from other existing middlewares.

3. MERMAID

3.1 Terminology

Throughout this paper, several software-related terms are used, such as *software architecture*, *middleware* and *framework*. Due to the fact that some of these terms are sometimes used with different meanings, we start by clarifying how they should be interpreted in the context of this paper.

The term *software architecture* is used in various contexts and there is no common definition of what it really means. Crispen & Stuckey define *software architecture* as (Crispen and Stuckey, n.d.):

"*An Architecture (...) consists of (a) a partitioning strategy and (b) a coordination strategy. The partitioning strategy leads to dividing the entire system in discrete, non-overlapping parts or components. The coordination strategy leads to explicitly defined interfaces between those parts*"

This is the general definition of software architecture that is used in this paper. Although this definition seems quite good, confusion may still arise between two different perspectives of a software architecture: horizontal software architecture and vertical software architecture. The difference between these two is that while a horizontal software architecture concerns only the organization and interaction of software components within a defined abstraction level, a vertical software architecture defines components and inter-component communication spanning several abstraction levels.

As an abstraction level we mean a collection of software entities that provide an abstraction for all underlying components in a system, be them other software entities or hardware. The lowest abstraction level is the computer's firmware, normally followed by assembly routines, the operating system and system libraries, as proposed in (Tanenbaum, 1979)

For the sake of simple terminology, the vertical software architecture will be called as the *system architecture* while the horizontal software architecture will be referred simply as the *software architecture*.

In addition to the given definitions, we can also define a software framework or simply *framework*: from the developer's point of view inside a given abstraction layer, a framework is a collection of software entities that reside in a lower abstraction layer and which are used in the current one. These software entities work cooperatively for a certain task in a certain domain. A framework exhibits a certain software architecture that will determine how other software entities will access/use it.

Finally, software middleware or simply *middleware* is defined as the collection of abstraction levels that are between the system libraries and end developer's programming abstraction level (hence the name). The end developer's abstraction level may be the final abstraction level or not. If not, it's up to the developer to build the remaining levels.

3.2 MeRMaID Entities

The most relevant entities 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 middleware favors a primitive 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., $\mathsf{see}(x)$, where x can be ball, pole, or field_line, in the soccer domain, or $\mathsf{near}(r,x)$, 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 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).

3.3 Current Implementation

Our high-level software architecture is divided in 3 major building blocks, each of them having several sub-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: handle the low level interface with physical-world devices (e.g. motors, sonars, cameras).
  - Sensors: obtain information from the devices (e.g. odometry, obstacle location, ball position)
  - Information Fusion: fuses information from several sensors (which can be sensors onboard the robot or from external sources)
  - Primitive Actions: see definition in previous section
  - Navigation Primitives: see definition in previous section
- WISDOM (i.e. a very relevant requirement for intelligence to be displayed): module acts as a central point of information storage and has the ability to generate events based on predicate changes
  - World Info: store general purpose high-level data, relevant for predicate evaluation. World Info may hold information originating from other robots.
  - Event Generator: generates events based on predicate changes.
- 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.

This description of MeRMaID's software architecture is intended to be a guide for the system developer. The developer is still free to choose the concrete implementation of each component. For instance, we currently have 3 different possible implementations for components inside CORTEX: Finite State Machines, Petri Net and Fuzzy Logic-based Behavior Arbitration. What this high-level software architecture defines is which components should exist and how should they interact with each other. In Figure 1 a diagram of the architecture is shown, with the relationships that are expected between components.

Underlying this high-level architecture, we have developed a low-level software architecture that we call *Support*. Issues such as communications and computing environment abstraction are handled at this level. We base our solution on the *Active Object* design pattern (Lavender and Schmidt, 1996).

All components (e.g., a primitive action) are implemented as Services that run inside Active Objects. Each Active Object provides a completely independent context and flow of execution. A suitable framework is supplied to the service developer in order to cope with communication and interaction with other Services.

We have developed a special object called ActiveObject. Objects of these kind are able to run services, so they have their own execution flow as well as execution context (i.e. ActiveObjects are completely independent from each other). With this kind of construction we are able to abstract completely from the underlying computing platform. Services do not (and should not) need to

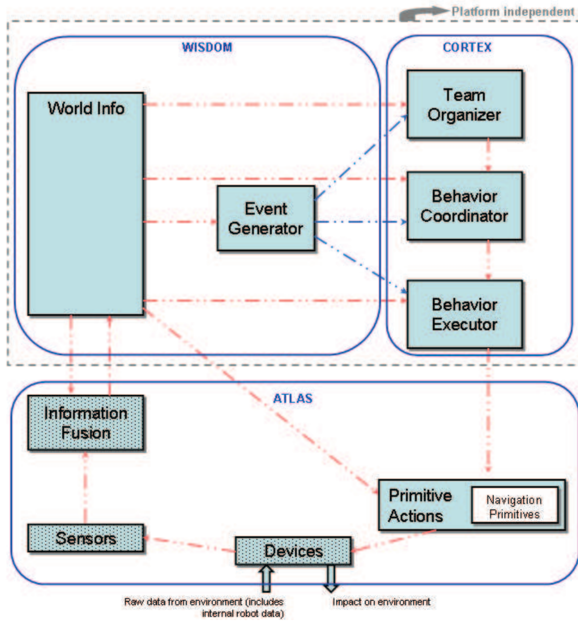

Figure 1. MeRMaID high-level software architecture block diagram.

know in which hardware platform they will run, they just know that they will run inside an ActiveObject.

All services derive from a common base class named Service. Every instance of Service that is running has a reference to an ActiveObject in which it executes. This way, a useful framework is supplied to the Service developer (in the form of methods of the ActiveObject and Service objects). With this framework, the developer can control how Services run inside an ActiveObject and are able to interact with other Services. Currently implemented methods for Services to interact with each other are:

- Asynchronous generic data communication
- Synchronous generic data communication
- Publish/Subscribe data diffusion (by name).

The ActiveObject control framework gives the developer control over how the Service will run. Start, pause, and stop operations are possible, as well as defining at which rate should the Service be regularly run (if any).

Having this kind of framework, we effectively hide the computational system from the developer. This way, the Service's code is platform independent, enabling the reuse of algorithms without needing to re-implement them. As long as a suitable implementation for a ActiveObject is built for the desired platform, all the previously developed Services should run and behave correctly.

4. APPLICATION EXAMPLE

MeRMaID is currently being used on real soccer robots in RoboCup Soccer Middle Size League. Some examples of MeRMaID entities in this domain were provided in the previous section. Here, we work out part of the components needed for an application where a (humanoid) robot operates in an urban environment to assist people in the street by guiding them to a specific location, though it could be required to act as a surveillance robot, when needed. The purpose is to illustrate how the different entities of our middleware work together to support the required robot behaviors. We assume that all the sensory and locomotory components of the robot have been developed under MeRMaID's framework (i.e., they were implemented as services running in AOs **sensor** and **primitive actions** in Figure 1) and are ready to use.

Finite State Machines are used for decision making at all three components of CORTEX, therefore the decision kernel system is purely event-based. The state machines for the Team Organizer, Behavior Coordinator and Behavior Executor are presented in Figure 2.

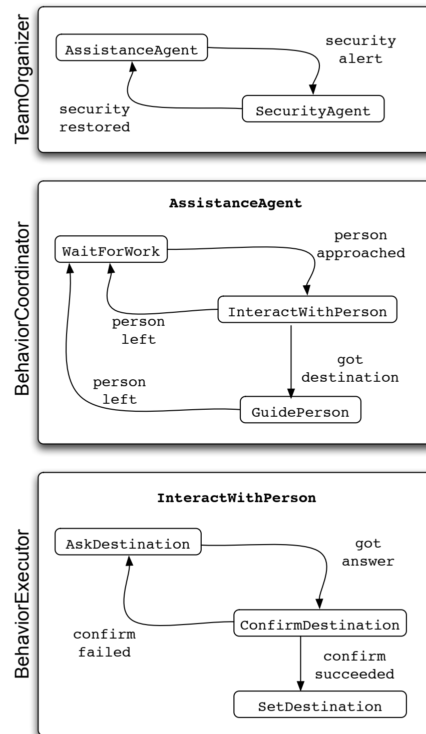

Figure 2. State machines of the CORTEX components for the human assistant aplication.

Events such as `security alert`, are triggered by wireless communication from other involved agents, such as a patrolling robot, or a human guard. Other events, such as `person approached` are triggered by the boolean value change of a

predicate stored in the **World Info**, such as `Far`(*Person*). Note that the information about the person distance to the robot is typically obtained locally by the robot, but could be the result of the fusion of the information provided by other robots in the team observing the person and running their own instances of MeRMaID. Information fusion is accomplished at the ATLAS block with the same name, and may rely either on the information communicated by several sensors in one robot and/or in several team members.

The set of roles, corresponding behaviors and primitive actions for each behavior are listed next in this order:

- `AssistanceAgent`
  - · `WaitForWork`
    - `StayPut`
  - · `InteractWithPerson`
    - `AskDestination`
    - `ConfirmDestination`
    - `SetDestination`
  - · `GuidePerson`
    - `Goto`(*waypoint*)
- `SecurityAgent`
  - · `RandomPatrol`
    - `Goto(randomPosition)`
    - `ScanArea(randomTime)`
  - · `Guard Area`
    - `Goto(areaExtremity)`
    - `ScanArea(shortTime)`
    - `Goto(otherAreaExtremity)`
  - · `Alarm+Pursue`
    - `IssueAlarm`
    - `Goto(nearSuspect)`

The state machines in Figure 2 illustrate how does the robot switch between the two available roles, how does it switch between the available behaviors for the `AssistanceAgent` role, and how does it switch between the primitive actions available for the `InteractWithPerson` behavior, when this behavior is running.

5. CONCLUSIONS AND ONGOING WORK

In this paper we described MeRMaID, a multiple-robot middleware that extends current robotic middleware by defining an entity set and a decision kernel which standardize the development of modules part of the multi-robot system. Our middleware meets the major requisites for multiple-robot middleware, such as parallelism, inter-component independency, data sharing, and servicing, as well as modularity, multi-paradigm interaction and real-time execution.

MeRMaID is written in C++ and runs currently under Linux OS. Ongoing work concerns its implementation in SONY's OPEN-R OS, so that it can control SONY AIBO robots as well, simultaneously showing an example of platform independence. We also intend to enhance our support level framework by providing communication with service-based semantics, localization transparency of Services and various code improvements to simplify the use of MeRMaID. Also in our plan list is the development of graphical user interfaces to support code development, execution and debug.

Acknowledgments This work was supported by the Fundação para a Ciência e a Tecnologia (ISR/IST pluriannual funding) through the POS_Conhecimento Program that includes FEDER funds.

References

Bruyninckx, H. (2001). Open robot control software: the orocos project. *Proceedings 2001 ICRA. IEEE International Conference* **3**, 2523– 2528.

Crispen, R. and L. Stuckey (n.d.). Structural model: Architecture for software designers. Boeing Defense & Space Group, P.O. Box 240002 M/S JM-70, Huntsville, Alabama 35824.

Lavender, R. Greg and Douglas C. Schmidt (1996). Active object: an object behavioral pattern for concurrent programming. pp. 483–499.

Lima, P., L. Custódio, M. Arroz, H. Costelha, B. Damas, C. Gil, G. Neto, P. Pinheiro and V. Pires (2003). Isocrob 2003 team description paper. *Proceedings of RoboCup 2003 Symposium*.

Metta, Giorgio, Paul Fitzpatrick and Lorenzo Natale (2006). Yarp: Yet another robot platform. *International Journal on Advanced Robotics Systems, Special Issue on Software Development and Integration in Robotics* **3**(1), 43–48.

Pirjanian, P. (1999). Behavior coordination mechanisms - state-of-the-art. Technical Report IRIS-99-375. Institute for Robotics and Intelligent Systems, University of Southern California. Los Angeles, CA, USA.

Tanenbaum, A. (1979). *Structured Computer Organization*. Prentice-Hall.

Utz, H., S. Sablatnog, Enderle and G. Kraetzschmar (2002). Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation* **18**(4), 493–497.