Multi-Robot Systems Middleware Applied to Soccer Robots

Nelson Ramos, Marco Barbosa, Pedro U. Lima
Instituto de Sistemas e Robótica, Instituto Superior Técnico
Av. Rovisco Pais, 1 – 1049-001 Lisboa, PORTUGAL

Abstract—This paper describes the basic concepts and features of a Robotic Middleware for multi-robot teams. The middleware puts constrains on some of the robot 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. We show examples of the middleware application to cooperative soccer robots.

I. 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 [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).

Several robotic-oriented middleware solutions exist, but they tend to offer very low level functionalities to the end developer (e.g., YARP [2] 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 [3] and OROCOS [4].

In this paper we introduce MrMaID (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 MrMaID, a high-level software architecture is presented to the developer, who must develop (or reuse) components that fit the architecture. This way, MrMaID ensures that separately developed components will be more easily assembled together at integration time. Moreover, MrMaID 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.

MrMaID is an evolution of previous versions of the software architecture of the ISR/IST RoboCup Soccer Middle Size League team ISocRob [5], 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: MrMaID’s components are detailed in Section II. In Section III a small tutorial introduction to Service development with MrMaID is given. Section IV 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 V.

II. MrMaID

A. MrMaID Entities

The most relevant entities in MrMaID 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., see(\(x\)) where \(x\) can be bull, 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 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\(_\text{object}\) changes its value from True to False, and vice-versa for event got\(_\text{object}\); event found\(_\text{object}\) occurs when the predicate see\(_\text{object}\) 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).

### B. 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.

![MeRMaID high-level software architecture block diagram](image)

**Fig. 1.** MeRMaID high-level software architecture block diagram.

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 [6].

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 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.

III. DEVELOPING A SERVICE IN MeRAMID

The basic low-level building block of MeRAMID is the Service. In the following subsections we will demonstrate how to use and develop a Service. We will start with a simple example showing a Service whose only task is to write periodically a message to a log file. Next we will show how to put that Service running by using an ActiveObject. Finally, we will demonstrate a mechanism for two Services to interact using asynchronous data communication.

The Service developer doesn’t need to know anything about the physical computational device in which the Services are going to run. The basic concepts that the developer needs to know are that, by extending the Service class, he/she can define how the Service is going to behave, and that, after registering the Service, he/she needs to create an ActiveObject in which the Service will run.

In Figure 2 we can see a UML class diagram resuming the parts of MeRAMID Support that we will use in the examples presented, and their relationships. We will create three new classes that inherit from Service: HelloService, ProducerService and ConsumerService. In these new classes we will only implement (and override, in C++ terms) the methods that are relevant for the Service’s behavior. Inside these overridden methods, the Service logic will be built.

A. Basic service development: HelloService

The HelloService is a simple Service whose task is to write periodically a message to a log file. Each Service has access to a log file where it can log messages relevant for information, warning or error reporting. Every Service has methods for its initialization and deinitialization (doStartup() and doShutdown() respectively). In doStartup() the developer is supposed to setup a Service so that it is ready for execution, while in doShutdown() the developer should free all resources gained in doStartup() and in the normal execution of the Service.

The serviceDefault() method is the method that is called by default when a Service is setup to run periodically. So, to implement HelloService we override this method, writing code that will execute the Service’s default task. In this case it is writing a message ("Hello Robot!") to the Service’s log file. All running instances of a Service have a log file in which they can write informational messages with the logInfo() message. Other methods exist for error and warning messages.
To set the Service to run periodically, we call inside the doStartup() method a method that controls this aspect of the Service: setServiceDefaultActive(). This method has two arguments, one stating if the period calling of serviceDefault() is active, which we set to True, the other stating how much times per second should serviceDefault be called, which we set to 1. This way, HelloService's serviceDefault() method will be executed once every second, writing "Hello Robot" to the log file.

```java
class HelloService : public Service {

    void doStartup() {
        setServiceDefaultActive(true, 1);
        /* order service to run periodically, 1 time per second*/
    }

    void serviceDefault() {
        logInfo("Hello Robot");
    }
}
```

B. Service deployment with an ActiveObject

Once we have the code for a Service written, we have to get it running inside an ActiveObject. That is done in 3 steps:

1) Register the Service in the ActiveObjectFactory.
2) Create an ActiveObject for the desired service.
3) Call the doStartup() and start() methods of the ActiveObject

After these 3 steps, the ActiveObject will be fully deployed. In our example, HelloService will be running it's serviceDefault() method periodically once a second.

```java
int main() {
    ActiveObjectFactory::registerService(  
        "HelloService",  
        &HelloService::myFactory);  

    ActiveObject * helloAO =  
        ActiveObjectFactory::createActiveObject(  
            "HelloService",  
            "HelloService");  

    helloAO->doStartup();
    helloAO->start();

    ActiveObjectManager::waitForAllActiveObjects();  
    // waits for all ActiveObjects to terminate their execution */
    helloAO->doShutdown();
}
```

C. Service interaction with asynchronous data communication

Now that we have shown how to build and deploy a Service, we'll show one of the available methods for Services to interact with each other: asynchronous data communication. ProducerService is very similar to HelloService, but instead of writing a message to the log file, it sends a message to the ConsumerService. This is done with the sendMsg() method, which accepts as arguments the name of the Service where the message should be delivered and the data to be delivered in the form of what we call a ParameterList. This ParameterList acts as a structure that associates data to names. The data can be of any of the several datatypes that McRMaID offers to developers (e.g. Boolean, Float, String, Posture, Obstacle, etc.). McRMaID's Support handles all the issues relating to sending the message to its receiver, so that the developer doesn’t have to handle with all the details of communication protocols.

```java
class ProducerService : public Service {

    void doStartup() {
        setServiceDefaultActive(true, 1);
        /* order service to run periodically, 1 time per second*/
    }

    void serviceDefault() {
        ParameterList p;

        p.addParameter("message",  
                       new String("Hello Robot"));  
        sendMsg("consumerService", p);
    }
}
```

ConsumerService only processes messages that it receives and, if they are well formatted, writes their contents to the log file. Message processing is done by overriding the processMsg() method. This is the method called every time a Service receives a message. Having received the message, it is up to the developer to check its name/data pairs and do the suitable action. In this example, the parameter "message" is asked for from the ParameterList. If data with this name exists it is printed to the log file, otherwise, a message indicating that the message received has an unknown structure is printed. The ConsumerService does not need to know who sent the message, only that it received a message and should process it according to its function.

```java
class ConsumerService : public Service {

    void processMsg(ParameterList params) {
        Parameter * p =  
            params.getParameter("message");

        if (p != NULL) {
            logInfo("Received message: " +  
                    new String(p)->getString());
        } else {
            logInfo("Received unknown message");
        }
    }
}
```

Just like the HelloService example, it is necessary to deploy the Services. This is done the same way, by registering the Service, creating the ActiveObject and calling the doStartup() and start() methods.
```c
int main() {
    ActiveObjectFactory::registerService("ProducerService", &ProducerService::myFactory);
    ActiveObjectFactory::registerService("ConsumerService", &ConsumerService::myFactory);

    ActiveObject * consumerAO =
        ActiveObjectFactory::createActiveObject("ConsumerService", "consumerService");

    ActiveObject * producerAO =
        ActiveObjectFactory::createActiveObject("ProducerService", "ProducerService");

    producerAO->doStartup();
    producerAO->start();

    consumerAO->doStartup();
    consumerAO->start();

    ActiveObjectManager::waitForAllActiveObjects();
    /* waits for all ActiveObjects to terminate their execution */

    producerAO->doShutdown();
    consumerAO->doShutdown();
}
```

IV. APPLICATION EXAMPLE

MeRMAID is currently being used on real soccer robots in RoboCup Soccer Middle Size League.

In this section we describe, with a real example, how a robot programmer should implement behaviours in robots with MeRMAID. CORTEX, as illustrated in section II, is divided in 3 component entities - TeamOrganizer, BehaviorCoordinator and BehaviorExecutor - corresponding to 3 levels of hierarchic decision. Any of these 3 entities can be implemented independently, using a formalism chosen by the developer, e.g., finite state machines, Petri nets, or fuzzy logic-based behaviour arbitration. Each of CORTEX’s components is implemented as a Service. The selection of the PrimitiveActions to execute is done by the chained decision of these 3 entities. MeRMAID guarantees that this mechanism is independent of the particular implementation of CORTEX’s components, by defining each Service’s interface, stating clearly which messages each Service should be able to receive and send. This way, every implementation of TeamOrganizer has to select a Role, every implementation of BehaviorCoordinator has to select a Behavior and every implementation of BehaviorExecutor has to select a PrimitiveAction.

We’ll present a simple example with all components running finite state machines.

Soccer playing robots can exhibit various kinds of individual, relational and organizational behaviors. The example of Figure 3 models an organizational behaviour defined with two Roles: Attacker and Supporter. The team of robots can have as much robots as desired. A Role is assigned dynamically to each of the robots, according to the control logic implemented by TeamOrganizer. In this example we will assign the Roles based on the following heuristic: the robot who is closest to the ball is in better condition to be the Attacker. With this heuristic we define two events: GotClosestToBall and LostClosestToBall which will occur when the predicate ClosestToBall changes from False to True and vice-versa, respectively. So, when a robot which is currently in the role Supporter gets the event GotClosestToBall it will change its role to Attacker. The robot which was previously with the Role Attacker, will receive the LostClosestToBall event and change its role to Supporter.

![Attacker and Supporter](image)

**Fig. 3. Team Organizer state machine for the soccer robots**

In Figure 4 we opted to show an example we chose to implement the Attacker Role, because it is more interesting than the Supporter Role. At this level inside CORTEX, it is decided, continuously, which is the best Behavior to execute so that the robot will best fulfill its Attacker Role. In this case, the BehaviorCoordinator has to worry about two distinct issues: the command events coming from the referee box should be obeyed according to the robotic soccer rules and, at the same time, the behaviour should be chosen as the one being the best to cope with the current game situation (taking into account the position of the robot, the ball, the opponents and the robot’s teammates).

To handle referee-imposed game situations, the following events exist:

- **GameStopped** occurs when the referee orders the game to stop while it is proceeding normally
- **GameStarted** occurs when the referee orders the game to continue while it is stopped
- **FoulCommitted** Occurs when the referee, after stopping the game, says that it was stopped because of a foul committed by the robot or one of the robot’s teammates
- **FoulSuffered** occurs when the referee, after stopping the game, says that it was stopped because of a foul committed by one of the opposing robots

These events are received directly from the referee, while the follow are based on Predicates:

- **FoulTaken** occurs when, while the game is stopped and the opposing team was told to take a foul that was committed in its favor, the opposing team’s robots have already taken the foul
- **FoulTimeExpired** occurs when, while the game is
stopped and the opposing team was told to take a foul that was committed in its favor, the opposing team's robots have taken too much time to take the foul.

During normal game play and without the referee's intervention there are two other possible events:

- **SupporterGotGoodPosition** occurs when, while in normal game play, a robot running the Supporter Role is in a favourable position
- **SupporterLostGoodPosition** occurs when, while in normal game play, a robot running the Supporter Role is not anymore in a favourable position

These events are based on the SupporterInGoodPosition predicate.

![Attacker state machine for the soccer robots](image)

**Fig. 4.** Attacker state machine for the soccer robots

In Figure 5 the IndividualAttack Behavior is illustrated. This finite state machine chooses Primitive Actions (each state having an associated Primitive Action) and runs inside BehaviorExecutor (when IndividualAttack is activated by BehaviorCoordinator). Since there is no referee intervention at this level, all events are based on predicates. These predicates where chosen as the pre-conditions necessary for Primitive Actions to execute. The finite state machine is built in such a way so that it is a chain of Primitive Actions in which each Primitive Action executes actions that take the robot to a state that meets the pre-conditions necessary to successfully run the next Primitive Action (being the pre-conditions assessed by Predicates).

To build these finite state machines, the developer may use a graphical interface which generates XML files containing a suitable representation of the finite state machines that will run in the robot. If all the necessary Primitive Actions and Events are already implemented, the XML file is all that is necessary to model the robot's behavior, with no need to write extra code.

V. CONCLUSIONS AND ONGOING WORK

In this paper we described McRMaID, 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.

McRMaID 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 McRMaID.

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 plurianual funding) through the POS.Conhecimento Program that includes FEDER funds.

REFERENCES