# Programming a multi-agent system with MASL

Dominique Duhaut, Yann Le Guyadec, Michel Dubois

Proceedings of the 2008 IEEE/ASME
International Conference on Advanced Intelligent Mechatronics
July 2 - 5, 2008, Xi'an, China

# Programming a multi-agent system with MASL

Dominique Duhaut, Yann Le Guyadec, Michel Dubois
*Valoria*
*Université de Bretagne Sud*
*Lorient Vannes, Morbihan, France*
dominique.duhaut@univ-ubs.fr

***Abstract* - Expressing the general behaviour of a set of robots working together is a difficult task. Self reconfigurable robots or a team of robots playing football are examples of such problem. To control a set of homogeneous or heterogeneous robotic components, one needs to express synchronous and asynchronous computation, from a local or a global point of view. In this paper we propose a unique language to express the behaviour of a set of heterogeneous robots.**

***Index Terms* - multi-agent systems, self-reconfigurable robots, RoboCup, programming language.**

## I. INTRODUCTION

Robot programming is a difficult sport that has been studied for many years [13]. This particular field often covers some very different concepts such as methods or algorithms (planning, trajectory generation...), and typically, architectures which are usually hierarchical, and for robot control: centralized [1], reactive [8], hybrid [2, 9, 16]. Therefore, languages are developed to implement these high-level considerations [17, 21]. Different approaches have appeared through functional [3, 4, 12] deliberative or declarative [5, 16, 18], synchronous characteristics [17]. Nonetheless, the difficulties of robot programming can by schematically summarized by two main characteristics:

- One is that the programming of elementary action (primitive) on a robot is often (even always) a program including many processes running in parallel with real-time constraints and local synchronization.
- The other is that in its interaction with the environment the program running the robot (sequence of primitives) must be able to carry out traditional features: interruption of event or exception and synchronization with another element.

The recent introduction of teams of robot, where cooperation and coordination are needed, introduces an additional difficulty which is that the programming is no longer reduced to a single physical system. The problem is then to program the behaviour of a group of robots or even of a society of robots [10, 11, 14]. In this case (except in the case of centralized control) programming implies loading, in each robot, a program which is not necessarily identical to the others because of the characteristics of the robots: different hardware, different behaviour and different programming languages. These various codes must in general be synchronized to carry out group missions (foraging, patrol movement...) and to have capacities of reconfiguration according to a map of communication cooperation.

From the human point of view it is then difficult to have simultaneously an overall description of the group of robots.

We met this problem in two types of distinct applications: RoboCup and the self reconfigurable robots [6, 7, 20]. In Robocup the teams of robots play football [15, 19] and the players/robots which are on the ground have different types of behaviour according to the dynamics of the environment. It is thus necessary to be able to express when and how a player playing offence decides to play defence (and vice versa). Here we look for the "re-evaluation" of the behaviour. For self reconfigurable robots another problem is that the walking motion implies "synchronisation" in the movement of the robot components. Then, we need to express that all the robots participating in the movement carry out their actions at the same time. We have noted that the traditional languages, which could be used, did not provide simple constructions to address this double problem: the expression of an attempt at "reevalutation" of its behaviour, the nature of the parallel execution of the group of robots.

The main idea of our work is thus to give a formal definition of a general language, MASL (multi-agent system language), to express six properties:

- Heterogeneous agents,
- Parallelism synchronous asynchronous,
- Communication variables, events,
- Dynamic integration of agent
- Message passing synchronous asynchronous
- Permeability dynamic

In this paper, section II will give a quick definition of the six properties. While section III will present some examples of the MASL language to show how it solves these properties.

## II. SIX PROPERTIES

To the six following properties we have to add that MASL also respects scalability constraints. This means that the following is independent of the number of agents instanced in the MASL code. First, the six properties must be defined.

### A. *Heterogeneous agents*

The program of an autonomous robot is an "agent" in the sense of agent programming. Robots are different. They may be identical at the beginning but become different due to their dynamics or may be different by construction (leg, wheel, manipulator …) and can be running different operating systems with different local programming languages.

MASL addresses all these differences by introducing an abstraction of the robot in which the capabilities of the robots are described. This abstraction will define a "type" of the MASL language. This type will be instanced to declare a corresponding agent.

### B. *Parallelism synchronous asynchronous*

A set of robots running their code is usually described as an asynchronous model of execution. This means that all the robots run their code at their own speed. But for specific tasks, the team of robots can be asked to execute their code synchronously. Then, after the execution of each instruction, the robot will wait for the others to finish their instruction. An example in human life would be dancing to music where everyone is following his own sequence of movement but everybody is doing it at the same rhythm.
MASL will integrate two descriptions of the execution of a sequence of code: synchonous and asynchonous

### C. *Communication variables, events*

This part is very classical but needed by the language. It defines the possibility of a set of robots to share variables or events. MASL offers three levels for sharing information: the whole set of robots (global variable), restricted to an agent (local level) and an intermediary level called "group" level in which a specific set of robots can access a piece of information. This set changes dynamically depending on the section of the code running.

### D. *Dynamic integration of agent*

By this property we mean that an agent running its code with a group (for instance playing offence on a football team) will be able to change its affiliation and become a member of another group (for instance defence). MASL authorises an agent to quit a group and to join another one

### E. *Message passing synchronous asynchronous*

When an agent asks another one for a service, there are two ways of managing the dynamics. First, the caller is blocked until the service is delivered by the callee, or the caller receives the result of the service later, but is free to work during the execution of the service. The first call is synchronous (caller blocked) the second one is asynchronous (the result will be given in the future). MASL will define these two types of message passing.

### F. *Permeability dynamic*

This original notion is defined to express that an agent cannot always execute its entire set of primitive capabilities. For instance, if the communication is not working then it is not possible to ask for a service. At another time, the agent can communicate, but due to his position in the environment he must be silent (children sleeping in the room). The permeability will define a set of states of the agent and will provide some instruction to manipulate it. These states define the number of services available from the agent.

## III. MASL EXAMPLES

In this section, agent will be used to indicate the program running the primitives of the robot.

### A. *Heterogeneous agents*

The objective is to program a set of heterogeneous agents working together. Then the proposed approach is to import the list of capabilities of agents from an XML description.

Example:

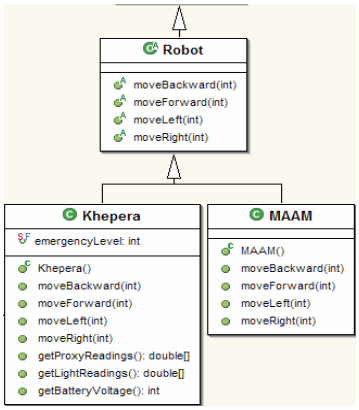


Figure 1: XML description of the robot primitves

An XML file describes the Khepera robot and MAAM robot. A set of primitives is defined for each of them.

```
01| import Khepera.xml as Khepera;
02| import MAAM.xml    as MAAM;

03| Khepera k1,k2   = newAgent(Khepera);
04| MAAM    m1,m2,m3 = newAgent(MAAM);
```

Here lines 01, 02 define a MASL type defined by the description of the XML file. Then, lines 03, 04 are the instantiations of 5 agents k1 and k2 of the Khepera type and m1, m2, m3 of the MAAM type.

```
05| asynchronous entry main (true) {
```

```
06|    .moveLeft(30);
07|    .moveForward(10);
08|    .moveRight(30);
09|    .moveBackward(10);
10| }
```

The `main` is executed by the 5 agents. The semantic of `.moveLeft(30);` is a self execution of the instruction which is the same as the Java **`this`**`.moveLeft(30);`. Each robot executes its own code independently in this first example. From that point in time, it is not possible to predict the order of execution of these instructions over the 5 agents.

*B. Parallelism synchronous asynchronous*

The previous example is an asynchronous execution in which all the agents execute their code independently.

```
05| synchronous entry main (true) {
06|    .moveLeft(30);
07|    .moveForward(10);
08|    .moveRight(30);
09|    .moveBackward(10);
10| }
```

Here the difference is the `synchronous` specification of the `entry main`. The `synchronous` specification means that after the execution of each instruction, all the agents inside the `entry` (here all agents because it is the main) will wait for the end of the execution of all the others.

In this case the movement of all the robots are made together. Once again, at that point in time, it is not possible to predict exactly the schedule table of the execution because one agent may be longer in carrying out its action and in which case all the others will be waiting.

The notion of entry is also used to define a section of code to be executed by a subset of agent. For instance

```
05|asynchronous entry example(.isMAAM()) {
06|    .moveLeft(30);
07|    .moveForward(10);
09| }
```

defines an instruction **`entry`** named `example` with a test `(.isMAAM())`. The evaluation of this test `(.isMAAM())` will select the agent authorized to execute the line sequence 6 ,7,8. The agent which does not satisfy the test will move to the next instruction (line 10). The instruction is used to form groups of agents. In this section it is possible to describe the behaviour, the subset, here move left or move forward.

MASL also proposes a

```
01| scalar entry e1( test)
```

to define an entry in which only one agent is allowed to enter. The first agent satisfying the `test` will enter in the `entry` and lock this `entry` so the others will skip this `entry`.
The notion of `entry` can be compared to the entry/accept of Ada language.

*C. Communication variables, events*

The visibility of variables or events depends on the place where they are defined and the statue expressed.

```
01|asynchronous entry main (true) {
02| shared int sglobalvar=0;
03| synchronous entry e1 (.isMAAM()) {
04|   shared int svar=0;
05|   local  int lvar=0;
06|   lvar++;
07|   svar++;
08|   .log("lvar="+lvar+"\n");
09|   .log("svar="+svar+"\n");
10| }
11|}
```

In this example, all the agents entering the main will share the variable declared in line 2 `sglobalvar` . This means that only one `sglobalvar` exists in the run-time and any modification from any agent will change its value.
Line 03 is an entry where only a subset of robot are selected (`.isMAAM()`). Thus, the variable `svar` defined in line 04 will be shared only for the 3 agents in this section of code. The variable `lvar` defined in line 05 (so 3 different `lvar` will be defined one per agent) instanced locally in each agent in this section of code.
In line 06, each agent will increment its internal variable `lvar`. Therefore, the final value written in line 08 will be 1 (3 times if there are 3 MAAM-type agents). But the `svar` final value will be 3 because each of the 3 MAAM agents will increment its value. Notice that this value will be written 3 times due to the synchronous scheduling of the **`entry`** `e1`.

On the same principle it is possible to define an event. This event can be **global** to all the agents if defined in the main or restricted to a **group** if defined in an entry or event **local** is visible only by the agent itself. The only instruction with events is `emit (event)`. The principle is based on the model of exception in Java.

```
01| asynchronous entry main (true) {
02| shared event sevent;
03| asynchronous entry e1 (.MAAM()) {
04|          …
06|   react (sevent) {
07|   .log("Reaction to an event");
08|   }
```

Here, line 02 is the global declaration of the **event** named `sevent` in the **entry** `main`. The agent entering the **entry** `e1` will execute the code (line 04). If during its execution, the `sevent` is emitted by any agent in some other section of the code, then the normal execution stops and jumps to the section **react**. Here (like catch in java) the event `sevent` will be searched in the list and the corresponding code will be executed.
If the code (line 08) includes the MASL instruction **resume** then the agent jumps back where it was in the code (line 04) when the event was emitted. This construction allows an agent to respond to an event and go back to the ongoing work.

*D. Dynamic integration of agent*

Here the problem is to allow an agent to quit a group to join another. To quit the group we can finish the normal execution of an entry using the instruction **break** or use the instruction **reelect** that comes back to the last entry and checks the test again. For instance, if in the previous example the line 08 is **reelect** then the agent will test the **entry** `e1` in line 03 again. In fact, it will test if the agent is still a MAAM robot or not. If yes, it will enter the **entry** `e1` again, or it will go to the next instruction (line 09).
This construction is useful to extract one agent from a group. However, the problem is then the adding of the agent to another group. We must thus imagine that it will find another **entry** in which the test will be true.

Notice that we also defined some instructions to lock or unlock an **entry**. This allows some agents to control the number of agents entering an **entry** section.

*E. Message passing synchronous asynchronous*

In the XML definition of the Khepera, the primitive moveLeft is defined. This can be used in two ways.
First, an agent k1 wants to move left, its code will then include:

```
01 : .moveleft(30);
```

expressing that the instruction is applied to agent k1 itself.

Or the code of agent k1 contains:

```
01 : k2.moveleft(30);
```

then k1 is asks agent k2 to move left. In this case we can imagine two scenarios.

- The execution of k1 is blocked until the move-left execution of agent k2 is done.
- The execution of k1 can continue during the execution of the move left of k2.

This depends on the definition of the primitive action `.moveleft()`.

The XML file defining the services of the robot can describe two types of primitives depending on the value of the field `synchronous_call`. When it is true, k1 is blocked and it is called synchronous message passing.

The problems are with the other kind: asynchronous message passing. The problem is that if you ask for a function producing an integer like `getBatteryVoltage() : int` in the XML file of Khepera. Then the code of k1 could be

```
02 : li= k2.getBatteryVoltage ();
```

expressing that a local variable `li` of k1 will receive the value of the voltage of the battery of k2. In the case of an asynchronous call, k1 will continue its work. If at some location in the code, k1 wants to use the value `li` then it must be sure that the assignment of `li` is accomplished.

To solve this problem we introduce in MASL the synchronisation barrier inspired from Java named **Label**.

```
01 : Label llabel;
02 : llabel.li= k2.getBatteryVoltage ();
…
06 : if (isFinished(llabel)){…}
```

Here, line 01 declares a new label `llabel`. Line 02 attaches an asynchonous message passing instruction to this label `llabel`. It is then possible for line 06 to test the label `llabel` to know if the instruction attached to it is completed or not. Note that it is possible to attach more than one instruction from an agent to a label, as well as different instructions coming from different agents.

*F. Permeability dynamic*

The permeability notion is completely connected to the previous message passing notion. It is used to express that some agents might not be able to answer a primitive call at some time during the execution.

The permeability is defined in the XML file of the robot. It defines a set of states of the robot and the list of primitives that can be executed in each of these states.

For instance, a permeability state: `standard` would determine if it is possible for a MAAM agent to execute all the 4 primitives defined Figure 1. But in a second permeability state: `damage`, only the `moveForward(int)` primitive could be called. This state would correspond to a robot having some problems.

Moreover, for all permeability states the allowed primitives are protected from execution in respect with the level: global, group, local. This means that dependending on the permeability state a primitive can be executed: by all agents of

the main, only by the members of the group (in the same `entry`) or only by the agent itself.

The permeability state can dynamically be changed only by the agent itself by the execution of a specific MASL instruction **`setAcceptState(`**`string`**`),`** where `string` is the name of the permeability state.

The MASL language also provides a `wait(string)` instruction. This instruction is used to put an agent in a waiting mode for its activation by a call on one of the primitives visible in the permeability state defined by `string`.

## IV. SIMPLE EXAMPLE

To show an example of the MASL language, we propose here a small sample in which we will distinguish groups of robots: one is an attacker and the second is a defender and one is the coach

```
05| asynchronous entry main (true) {
06| shared event mvBack, mvForward, move;
07| scalar entry coach (true) {
08|   local  int lv, li=0;
09|   loop
10|    lv=.analyseSituation(); li++;
11|    if (lv<0) emit mvBack;
       else emit mvForward;
12|    if (li==100) {li=0; emit move;};
13|   endloop
14| }
15| asynchronous entry attack(.isFast()) {
16|   loop .playAttack(); endloop;
17|   react(mvForward)
        {.moveForward(20);resume;};
18|   react(mvBack)
        {.moveBackward(20);resume;};
19| }
20| asynchronous entry defense (true) {
21|   loop .playDefense(); endloop;
22|   react(mvForward)
        {.moveForward(5);resume;};
23|   react(mvBack)
        {.moveBackward(5);resume;};
24|   react(move)
        {.setRandomFast();reelect(2);};
25| }
```

This example is built to show some features:
- definition of groups of robots,
- scalability,
- dynamic change of group,
- how to control a group from a supervisor.

The instantiation of the agents is not shown here. We assume that it is a set of agents for identical robots. All these agents will share 3 events (line 6).

During the execution the first agent to execute line 7 will become the coach (because it is a `scalar entry` only one can enter). The next agents will skip instruction line 7 and move to execute line 15. If the test performed on themselves (`.isFast()`) is true, then they will enter and go to line 16 to play in attack mode, the other ones will move to line 20 where they will enter (because the test is `true`) and run line 21 to play in defence mode.

We can see here that the initial group of agents is separated into 3 groups: one (alone) in the `entry coach`, a set of agents in the `entry attack` and the rest in the `entry defense`. Notice that it is independent of the number of robots.

Now the dynamic evolution will develop through the coach's behaviour. He analyses the situation (line 10) and decides what the two groups of attackers and defenders will do. So, in line 11 he emits the event `mvBack` (resp. `mvForward`). This event is global to all agents and they can `react` to it.
The attackers will `react` (line 17 (resp. 18) by a `moveForward`(20) (resp. `moveBackward`(20)) and then go back to their offence behaviour in line 16 when executing `resume`.
The defenders will `react` (line 22 (resp. 23) by a `moveForward`(5) (resp. `moveBackward`(5)) and then go back to their attacking behavior line 21 when executing `resume`.

We can see here how one group is asked to make big amplitudes (20), while the other one is not (5).

The coach can also force defenders to become attackers. Every 100 loops (line 12) the coach emits an `event move`. In this `event` only the defenders `react` (line 24). Their reaction is to randomly decide if they are `Fast` or not. After which they will reenter the program line 5 by the execution of the `reelect(2)` instruction. They will enter again in the `main`. Because the `coach entry` is locked they will move to line 15 to see is they become attackers (if yes, they enter line 16) or not (they move to line 20) and become defenders again.

Here we can see all groups of agents reevaluating their behaviour. It is, of course, possible in MASL to refine to a specific agent.

## V. SEQUENCE

In this section, we present the basic algorithm of the execution of a MASL instruction. It is divided into 4 steps:

1- Execution of the instruction, it is the execution of the primitive by the agent

2- Ask MASL runtime for the list of events. At this level the call to the basic primitive is finished and the agent looks if there is some events emitted. If so, it will jump to the `react` part of the code and search for the first instruction to execute, then go back to step one

3- Ask MASL runtime for the list of primitive calls. Here, the agent according to the permeability state will execute the primitive calls. The calls that are not allowed due to the permeability will be neglected.

4- Wait for synchronization. If the agent is in a synchronous entry then it will wait for the end of the group before looping to step1.

## VI. CONCLUSION

The MASL language proposed here allows for the description of multi-agents, multi-robot behaviour at three different levels: global, group, and local. The originality of MASL is the definition of the instruction entry to create groups of robots. This instruction can run dynamically in two modes: asynchronous or synchronous. Another particularity of MASL is the message passing construction which allows asynchronous or synchronous calls to be defined under a permeability state.
From MASL it is possible to define a rewriting algorithm to produce a source code for a specific robot programming language. This algorithm is under development.

The full description of the MASL language is available on:
**http://www-valoria.univ-ubs.fr/Dominique.Duhaut/MASL**


## ACKNOWLEDGMENT

This project is supported by the Robea project of the CNRS. All references to people participating in this work can be found in [22].


## REFERENCES


[1] J. S. Albus & al. NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM). NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, MD, 1987.

[2] R. Alur & al., *"Hierarchical Hybrid Modeling of Embedded Systems."* Proceedings of EMSOFT'01: First Workshop on Embedded Software, October 8-10, 2001

[3] J. Armstrong "The development in Erlang", ACM sighpla international conference on functional programming p 196-203. 1997

[4] M.S. Atkin & al. "HAC : a unified view of reactive and deliberative activity. Notes of the European conf on artificial intelligence 1999

[5] M. Dastani & L. van der Torre " Programming Boid-Plan agents deliberating about conflicts along defeasible mental attitudes and plans" AAMAS 2003

[6] C. Gueganno and D. Duhaut "A hardware/software architecture for the control of self reconfigurable robots" DARS 04 7th symposium on distributed autonomous robotics systems, June 23-25, Toulouse France.

[7] M. Jorgensen & all "Modular ATRON: modules for a self-reconfigurable robot" IEEE/RSJ int conf on intelligent robots and systems IROS 2004 Sendai Japan

[8] P. Hudak & al. " Arrows, robots, and functional reactive programming" lecture note in computer scinece 159-187 Spinger Verlag 2002

[9] F. F. Ingrand & al. "PRS : a high level supervision and control language for autonomous mobile robots", IEE int cong on robotics and automation Minneapolis, 1996

[10] E. Klavins "A formal model of a multi-robot control and communication task" IEEE conf on decision and control, 2003

[11] E. Klavins "A language for modeling and programming cooperative control systems" Int conf on robotics and automation ICRA 2004

[12] G. King "Tapir: the evolution of an agent control language" American association of artificial intelligence 2002.

[13] T. Lozano-Perez & R. Brooks "An approach to automatic robot programming" Proceedings of the 1986 ACM fourteenth annual conf on computer sciences 1986, ACM Press

[14] D.C. Mackenzie & R. Arkin "Multiagent mission specification and execution" Autonomous robot vol 1 num 25 1997

[15] F. Mondada & al. "Swarm–bot : for concept to implementation", IEEE/RSJ int conf on intelligent robots and systems IROS 2003

[16] D. Paul Benjamin & al. " Integrating perception, language an problem solving in a cognitive agent for mobile robot" AAMAS'04 July 19-23 2004, New York

[17] I. Pembeci & G. Hager "A comparative review of robot programming languages" report CIRL – Johns Hopkins University August 14, 2001

[18] J. Peterson & al. "A language for declarative robotic programming" Int conf on robotics and automation ICRA 1999

[19] T. Vu & al. "Monad: a flexible architecture for multi-agent control" AAMAS'03

[20] E.Yoshida & al. "Planning behaviors of modular robots with coherent structure issing randomized method" DARS 04 7th symposium on distributed autonomous robotics systems, June 23-25, Toulouse France.

[21] C. Zielinski "Programming and control of multi-robot systems" Conf. On control and automation robotics and vision ICRARCV'2000 Dec. 5-8 200à, Singapore

[22] http://www.univ-ubs.fr/valoria/duhaut/maam.