MASL: a Language for Multi-Agent System

Michel Dubois, Yann Le Guyadec, Dominique Duhaut

12

MASL: a Language for Multi-Agent System

Michel Dubois, Yann Le Guyadec and Dominique Duhaut
University of South Britanny. Université Européenne de Bretagne
France

1. Introduction

Our work takes place in the field of Multi-Agent Modular Robotic System. We propose to mix several paradigms of computation to offer a high-level point of view to the programmer into a new language, namely MASL for Multi-Agent System Language. Expressivity of MASL is illustrated on an example applied to a fleet of robots.
A Multi-Robot System (MRS) can be characterized as a set of robots operating in the same environment that operate together to perform some global task. In this chapter, we regard MRS as a particular form of Multi-Agent System (MAS). The main differences between general MAS and MRS are:

- The fact that in MRS direct communication is based on dedicated physical devices, resulting in a much more expensive and unreliable solution to attain coordination with respect to MAS.
- The number of robots acting in the same environment is still quite limited with respect to the number of agents in a MAS.
- Agents architectures in a MAS are usually deliberative (Nilsson, 1984) (Sense-Represent-Plan-Act), reactive (Brooks, 1989; Hudak et al., 2002) (subsumption architecture) or hybrid (Alur et al., 2000; Ingrand et al., 1996; Benjamin et al., 2004) as shown in figure 1 while in MRS we consider only the last two. The pure sense-represent-plan-act architecture, which is used to realize a high level deliberative behaviour, is not currently used in MRS because of its intrinsic limits, while the behaviour-based and the hybrid architectures are quite common, especially when the robot is situated in a highly dynamic environment, where a quick reaction to a new input is very important, being the environment itself uncertain and unpredictable.

Robot programming is a difficult task that has been studied for many years (Lozano-Perez & Brooks, 1986). This particular field often covers some very different concepts such as methods or algorithms (planning, trajectory generation...). Therefore, languages are developed to implement these high-level considerations (Pembeci & Hager, 2001; Zielinski, 2000). Different approaches have appeared through functional (Armstrong, 1997; Atkin et al., 1999; King, 2002), deliberative or declarative (Dastani & van der Torre, 2003; Benjamin et al., 2004; Peterson et al., 1999), synchronous characteristics (Pembeci & Hager, 2001). Nonetheless, the difficulties of robot programming can by schematically summarized by two main characteristics:

- One is that programming a set of elementary actions (primitives) on a robot often leads to (if not always) a program including many processes running in parallel with real-time constraints and local synchronizations.

Source: Multiagent Systems, Book edited by: Salman Ahmed and Mohd Noh Karsiti,
ISBN 978-3-902613-51-6, pp. 426, February 2009, I-Tech, Vienna, Austria

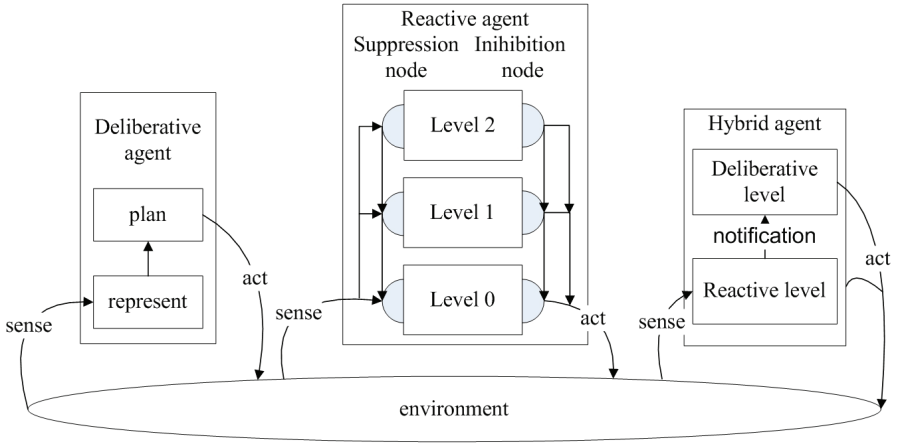

Fig. 1. Different types of agents in a MAS

- The other is that in its interactions with the environment, the program running the robot (i.e. the sequence of primitives) must be able to carry out traditional features: interruption of an event or exception and synchronization with another element.

The recent introduction of teams of robot, where cooperation and coordination are needed, presents an additional difficulty which is that the expression of computation is no longer limited 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 (Klavins, 2003; Klavins, 2004; Mackenzie & Arkin, 1997). 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 communicative cooperation.

From the human point of view it is then difficult to have a simultaneous overall description of the group of robots. We met this problem in two types of distinct applications: RoboCup and the self reconfigurable robots (Gueganno & Duhaut, 2004; Jorgensen et al., 2004; Yoshida et al., 2004).

Robotic soccer has been considered in the last years as an interesting test-bed for research in multi-agent and multi-robot cooperation. The uncertain dynamics and hostile environment in which the robots operate makes coordination of the multi-robot system a real challenging problem. While in the early years of the robotic league competitions the focus has been on improving the single robot capabilities, only recently coordination in the MRS has become a central issue. The different settings of each of the robotic leagues present several issues for coordination in MRS. In particular, in the Middle-Size league and the Four Legged league, all the robot sensors must be on board; therefore robots are more autonomous and have to deal with high uncertainty in reconstructing global information about the environment. On the other hand in the small size league the robotic agents can take advantage of a top view of the environment provided by a camera on the top of the field, therefore the coordination approaches in this league are mostly centralized. The use of coordination in the soccer domains has demonstrated significant improvement in the performance of the teams. In

Robocup the teams of robots play football (Vu et al., 2003) and the robots players 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 an offensive player decides to play defence (and vice versa). Here we look for the "re-evaluation" of the behaviour (new role in the team according to a given goal).
Self reconfigurable robots have been recently studied as swarm intelligence system using self organising and self-assembling capabilities shown by social insects and other animal societies (Mondada et al., 2003). One of the main difference between swarm-robotics and MAS is that MAS also supports deliberative or hybrid controllers for the robots rather than reactive ones. Swarm-robotics strictly discards deliberative and hybrid controllers because of its bias on simplicity. No robot in the swarm has a global knowledge of the environment or of the status of the swarm itself. Instead, each robot exploits only local information and a global behaviour emerges from the interactions among the individuals. Between each member of the swarm, direct communication makes use of some on board dedicated hardware devices, while indirect communication makes use of stigmergy (a mechanism of spontaneous, indirect coordination between agents or actions, where the trace left in the environment by an action stimulates the performance of a subsequent action, by the same or a different agent). The direct communication should be kept limited as much as possible and preferably communication should be done using broadcasting instead of using robot names or addresses or complex hierarchies based on robot addresses. 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 traditional languages did not provide simple constructions to address this double problem: the expression of an attempt at "re-evalutation" of its behaviour and the nature of the parallel execution of the group of robots.
The main idea of our work is thus to give a definition of a general language, MASL, in order to address 6 constraints:

- Heterogeneous agents,
- Composition of computation,
- Cooperation,
- Dynamic integration of agent,
- Permeability dynamic,
- Scalability.

In this chapter, section 2 will give a brief definition of the aforementioned six constraints, while section 3 will inspect related works in MAS for MRS and section 4 will present some examples of the MASL language to show how it solves these features. Section 5 will provide a simple example to show MASL expressivity. Implementation issues will follow and some future works will end.

2. Our model

First, the six constraints must be defined. And we present after the construction **`entry`** that unifies all these aspects.

2.1 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 …). They also 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 instantiated to declare a corresponding agent.

2.2 Synchronous/asynchronous computations

A set of robots running their code is usually described following an asynchronous model of execution. This means that all the robots run their code at their own speed. For specific tasks, the team of robots can be asked to execute their code synchronously: after the execution of each instruction, each robot waits for the others to finish their current instruction. An example in human life would be a group of people dancing on music in which each one is following one's own sequence of movement but all are doing it at the same rhythm.
MASL will integrate two descriptions of the parallel composition of sequences of code: synchronous and asynchronous.

2.3 Cooperation

Cooperation is usually made with communication variables, events and synchronous/asynchronous message passing.
Communication variables and events are 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 according to the section of the running code.
Synchronous/asynchronous message passing is used 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 exploits these two types of message passing.

2.4 Dynamic integration of agent

By this feature we mean that an agent running its code within 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 allows an agent to dynamically quit a group and to join another one.

2.5 Autonomy by dynamic permeability

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, an agent can communicate, but due to its position in the environment he must remain silent in some emergency cases. The permeability will define a set of roles of the agent and will provide some instruction to manipulate it. These roles define which services are activable in the set of all available services from the agent. The permeability of an agent refers to its autonomy. It can neglect access demands to its interface from other agents because it is not a passive object but an active one or a reactive one.

2.6 Scalability

MASL also respects scalability constraints. This means that the same controller can be applicated on variable sized group. Scalability is one of the desired characteristics of swarm robotics and swarm systems should work with large numbers of system components.
MASL language provides one construct named **`entry`** that unifies several paradigms of computation to offer a high-level point of view to the programmer.

3. Related works

In this section, we study the existing Multi-Agent Languages in MRS with respect to our six constraints. They are CHARON (Alur et al., 2000) (Coordinated control, Hierarchical design, Analysis, and Run-time mONitoring of hybrid systems), CCL (Klavins, 2003; Klavins, 2004) (Computation and Control Language), MRL (Nishiyama et al., 1998) (Multiagent Robot Language), Tapir (King, 2002), GOLOG (alGOl in LOGic) (Levesque et al., 1997), CDL (Mackenzie & Arkin, 1997) (Configuration Description Language) and XABSL (Lötzsch, 2004) (eXtensible Agent Behavior Specification Language).

	(1)	(2)	(3)	(4)	(5)	(6)
CHARON	+	Not so precise	SV, MP, E-	Not so easy	+	-
CCL	+	Not so precise	SV, MP	Not so easy	+	+
MRL	+	Not so precise	E, SV, MP		+	-
Tapir	+	Not so precise	MP, SV		+	-
GOLOG	+	Not so precise	E, SV		+	-
CDL	+	Not so precise	E, MP	-	-	+
XABCL	+	Not so precise	E, MP	Not so easy	-	+

Table 1. Existing Multi-Agent Languages Features

The legend is: SV (Shared Variable), MP (Message Passing), E (Events), E- (partially implemented), + (feature available), - (feature unavailable), nothing (not documented).

4. Informal semantics

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

4.1 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:

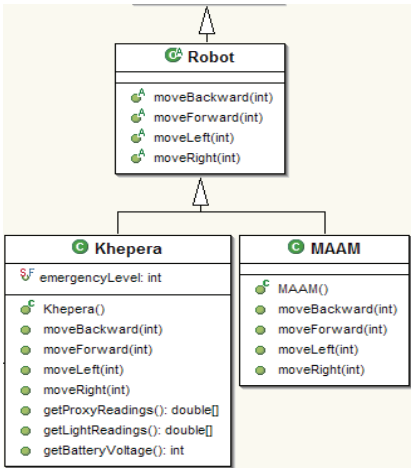

Fig. 2. UML class diagram of the robot primitives

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 referenced 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 (asynchronous entry). From that time, it is not possible to predict the order of execution of these instructions over the 5 agents.

4.2 Synchronous/asynchronous interleaving

The previous example is an asynchronous execution in which all the agents execute their code independently. In the next example, we study the synchronous version.

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

Here the difference is the `synchronous` keyword in the `main entry`. The `synchronous` keyword means that after the execution of each instruction, all agents inside the `entry` (here all agents because the test is `true`) will wait for the end of the execution of all others.
In this case the movement of all robots are performed simultaneously. Once again, at that point in time, it is not possible to predict exactly the schedule table of the execution because one agent may take longer in carrying out its action, in which case all others will be waiting.
The granularity of synchrony is macroscopic.
The notion of entry is also used to define a section of code to be executed by a subset of agents using an entry condition. 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 agents which don't satisfy the test jump to the next instruction (line 10). The instruction is used to form groups of agents. In this block it is possible to describe the behaviour, the subset, i.e 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.

4.3 Communication variables, events

The scope of variables or events depends on the place where they are defined and the local/shared modifier 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 a **local** event visible only by the agent itself. The only instruction with events is **emit** (event).

```
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 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 return to the ongoing work.

4.4 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 to add 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.

4.5 synchronous/asynchronous message passing

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 a 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 utility named Label, inspired from the Java interface `java.util.concurrent.Future`.

```
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 whether 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.

4.6 Dynamic permeability

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 in Figure 1. But for 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 virtue of the levels: global, group, local. This means that depending 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 be dynamically 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 until its activation by a call from one of the primitives visible in the permeability state defined by `string`. Them this agent behaves as a passive agent but constrained by this permeability state until its activation.

5. MASL expressivity in a RoboCup application

To show an example of the MASL language, we propose here a small sample in which we will distinguish three 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|   }
26| }
```

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 this 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 of the actions of the two groups of attackers and defenders. 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 whether 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 decide whether they become attackers (if yes, they enter line 16) or not (they move to line 20) or 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.

6. Toward centralized execution model

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 checks 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 step 1.

From agent architectural point of view, only some MASL features have a deliberative nature (shared variables for example). Broadcasted shared events have a reactive nature in contrast. MASL Controllers with shared variables will produce a deliberative loop. MASL Controllers without any deliberative feature will produce reactive loop. In this particular case, please replace in figure 3 hybrid/deliberative worlds by reactive one.
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 MASL to Java translator use XML files and MASL program to generate the reactive/deliberative loop in java. We plan also to develop the MASL to C++ translator.
We have studied some deployment cases and some different execution models shown in figure 3. Only the simulated MASL agents (in the far right) are working for now. The implementation of some MASL concepts (**`synchronous entry`**, broadcast of `events`, shared variables, **`react`**, **`resume`**, **`reelect`**) depends on the target execution model (centralized or distributed). Outside the remotely connected agent case, the MASL program

is replicated into each agent. In case of a distributed execution model, broadcasted events, access to (virtually) shared variables (physically replicated) and synchronisation mechanisms have to be implemented on top of the network. Consistency of access to shared variables has to be ensured in this case. We have defined features that are in all cases into each agent. They are managed by the local runtime. Other features are managed by centralized MASL runtime or distributed MASL runtime in respect to deployment case. The deliberative loop communicates only with local runtime that optionally uses shared services. This rule prevents to have to modify the MASL to Java translator nor the MASL to C++ translator to produce the source code for real or simulated agents for specific deployment case.

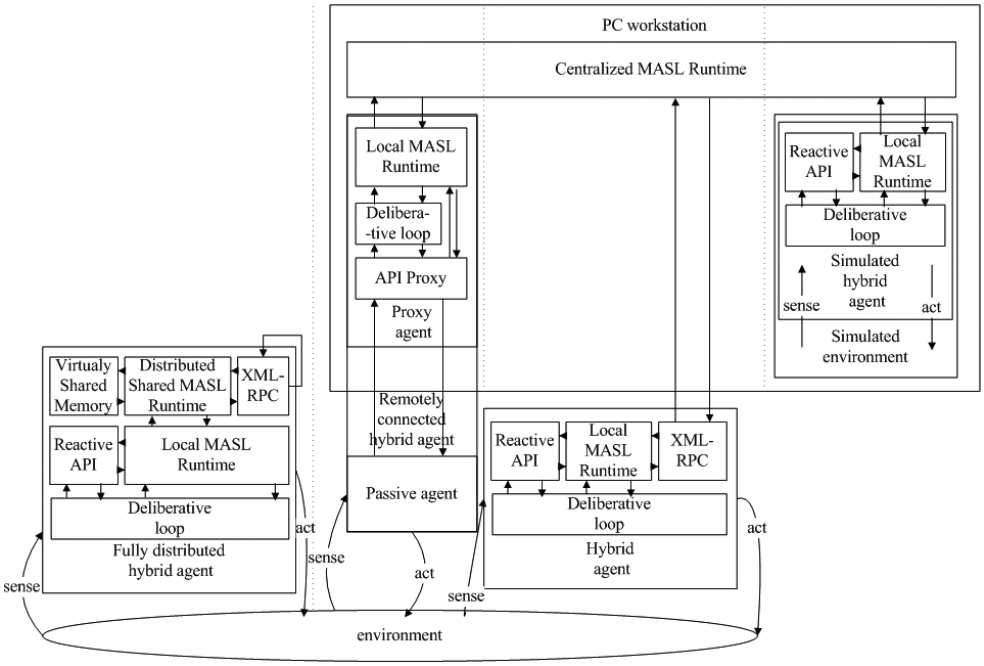

Fig. 3. Different deployment cases for MASL agents

Remote Procedure Calls (RPC) is a technique for constructing distributed, client-server based applications. It uses the notion of conventional local procedure calls, so the called procedure need not exist in the same address space as the calling procedure.

XML-RPC (de Rivera et al., 2005) is a specification and a set of implementations that extend RPC to allow procedure calls to be made over the Internet to machines with potentially different execution environments and operating systems. This makes use of HTTP as the transport and XML as the encoding. XML-RPC is designed to be simple, but also be powerful enough to allow complex data structures to be transmitted, processed, and returned. There are many XML-RPC implementations in various computer languages, e.g., C/C++, Java, Perl, and Python, and for various operating systems, e.g., GNU/Linux, Microsoft Windows, and Sun Solaris.

7. Conclusion and future works

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.
Future works are:

- We want to implement MASL to C++ translator,
- We want to test real robots controllers,
- We want to define some good principles in MASL programming,
- We want to provide GUI (Graphical User Interface) for MASL mission specification,
- We want to express the formal semantic to allow us to prove several properties in our controllers,
- We want to provide FIPA compliant services to our agents.

8. Acknowledgement

This work is a part of the MAAM project that is supported by the Robea project of the CNRS.

9. References

Alur, R.; Grosu, R.; Hur, Y.; Kumar, V. & Lee, I. (2000). Modular specification of hybrid systems in CHARON. *In HSCC*, pp. 6-19.

Armstrong, J. (1997). The development in Erlang, *ACM sighpla international conference on functional programming*, pp. 196-203.

Atkin, M.; Westbrook, D. & Cohen, P. (1999). HAC : a unified view of reactive and deliberative activity, *Notes of the European conf on artificial intelligence*.

Benjamin, P.; Lonsdale, D. & Lyons, D. (2004). Integrating perception, language an problem solving in a cognitive agent for mobile robot, *AAMAS'04*, July 19-23 2004, New York

Brooks R. (1991). Intelligence without Reason, *Proceedings of the IJCAI'91*, Sydney (Australie), Morgan-Kaufmann, pp. 569-595.

Dastani, M. & van der Torre, L. (2003). Programming Boid-Plan agents deliberating about conflicts along defeasible mental attitudes and plans, *AAMAS 2003*

Gueganno, C. & Duhaut, D. (2004). 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.

Hudak, P.; Courtney, A.; Nilsson, H. & Peterson, J. (2002). Arrows, robots, and functional reactive programming, *lecture note in computer science* 159-187 Springer Verlag 2002

Ingrand, F.; Chatila, R.; Alami, R. & Robert, F. (1996). PRS : a high level supervision and control language for autonomous mobile robots, *IEEE int cong on robotics and automation*, Minneapolis.

Jorgensen, M.W.; Ostergaard, E.H. & Hautop, H. (2004). Modular ATRON: modules for a self-reconfigurable robot, *IEEE/RSJ int conf on intelligent robots and systems IROS 2004* Sendai Japan

King, G. (2002). Tapir: the evolution of an agent control language, *American association of artificial intelligence*.

Klavins, E. (2003). A formal model of a multi-robot control and communication task, *IEEE conf on decision and control*.

Klavins, E. (2004). A language for modeling and programming cooperative control systems, *Int conf on robotics and automation ICRA*.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F. & Scherl, R. (1997). Golog: A logic programming language for dynamic domains, *Journal of Logic Programming*.

Lötzsch, M. (2004). XABSL - A Behavior Engineering System for Autonomous Agents. *Diploma thesis*. Humboldt-Universität zu Berlin.

Lozano-Perez, T. & Brooks, R. (1986). An approach to automatic robot programming, *Proceedings of the 1986 ACM fourteenth annual conf on computer sciences*, ACM Press

Mackenzie, D.C. & Arkin, R. (1997). Multiagent mission specification and execution, *Autonomous robot vol 1 num 25*.

Mondada, F.; Guignard, A.; Bonani, M.; Bar, D.; Lauria, M. & Floreano, D. (2003). Swarm–bot : for concept to implementation, *IEEE/RSJ int conf on intelligent robots and systems IROS*.

Nilsson, N. J. (1984). Shakey the robot. *Technical Report 323*, SRI Artificial Intelligence Center.

Nishiyama, H.; Ohwada, H. & Mizoguchi, F. (1998). A Multiagent Robot Language for Communication and Concurrency Control. *Proceedings of the International Conference on Multiagent Systems*, pp. 206–213.

Pembeci, I. & Hager, G. (2001). A comparative review of robot programming languages, *report CIRL* – Johns Hopkins University.

Peterson, J.; Hager, G.D. & Hudak, P. (1999). A language for declarative robotic programming, *Int conf on robotics and automation ICRA*.

de Rivera, G.G.; Ribalda, R.; Colas, J. & Garrido, J. (2005), A generic software platform for controlling collaborative robotic system using XML-RPC, *Advanced Intelligent Mechatronics. Proceedings, 2005 IEEE/ASME International Conference on*, Volume , Issue , 24-28 July 2005, pp. 1336 - 1341

Vu, T.; Go, J.; Kaminka, G.; Veloso, M. & Browning, B. (2003). Monad: a flexible architecture for multi-agent control, *AAMAS'03*.

Yoshida, E.; Kurokawa, H.; Kamimura, A.; Murata, S.; Tomita, K. & Kokaji, S. (2004). Planning behaviors of modular robots with coherent structure using randomized method, *DARS 04 7th symposium on distributed autonomous robotics systems*, June 23-25, Toulouse France.

Zielinski, C. (2000). Programming and control of multi-robot systems, *Conf. On control and automation robotics and vision ICRARCV'2000 Dec. 5-8*, Singapore.