Horocol language and Hardware modules for robots

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

Abstract

This work inserts in the general field of collective robotics. In this paper, we present the results on the design and the conception of (1) our robotics component called Atom, (2) the informal semantics of the HoRoCoL language. The expressivity of the language is illustrated on a simple example.

At the hardware level, we propose a versatile architecture easily adaptable for most mechatronic systems. The hardware is based on a processing unit developed around a CPU + FPGA computing system communicating through *bluetooth*. On this hardware we build a software architecture, where each robot embeds its own description in an XML file. Control interfaces or programming tools are self-reconfigurable, depending of the XML description of the robot. That enables quick technology transfer for many mechatronics applications.

At the software level, we present the Horocol language for programming a society or teams of robots. An example shows the principal features of the Horocol language. This language has been developed to offer a solution to express the behaviours of a set of teams of robots or agents. We focus on the originality of this language which is in the instructions for programming the team coordination.

Introduction

This project takes place in the more general field of reconfigurable modular robotics. We can mention several various experiments. The M-TRAN (Modular Transformer - AIST) described in [1], is a distributed self-reconfigurable system composed of homogeneous robotic modules. CONRO (Configurable Robot - USC), is a robot made of a set of connectible, autonomous and self-sufficient modules [2]. ATRON, is a lattice based self-reconfigurable robot [3], and also, PolyPod (Xeros) [4], I-Cube (CMU) [5], Hydra . These robots generally consist in modules working together and where each module is permanently linked to at least one other.

Programming such reconfigurable systems is a difficult task [6]. This field covers very different concepts like : methods or algorithms (planning, trajectory generation...), or classically, architectures for robot control, usually hierarchical : centralised [7], reactive [8], hybrid [9, 10, 11]. Some languages are developed in order to implement these high level concepts [12, 13]. Different paradigms are also proposed: functional [14, 15, 16], deliberative or declarative [17, 11, 18] and synchronous [12]. In any way, we can schematically summarise the difficulties of robot programming in two great characteristics:

- programming of elementary actions (primitives) on a robot is often a program including many process running in parallel with real-time constraints and local synchronisation
- interactions with the environment are driven via traditional features: interrupt on event or exception and synchronisation with another element.

The recent introduction of teams of robot, where cooperation and coordination are needed, introduces an additional difficulty : programming the behaviour of a group of robots or even a society of robots [19, 20, 21, 22]. In this case (except in the case of a centralised control) programming implies to load a specific program on to each robot because of the different characteristics of robots : different hardware, different behaviours and different programming languages. These distinct programs must in general be synchronized to carry out missions of group (foraging, displacement in patrol, ...) and have reconfiguration capabilities according to a map of cooperation communication.

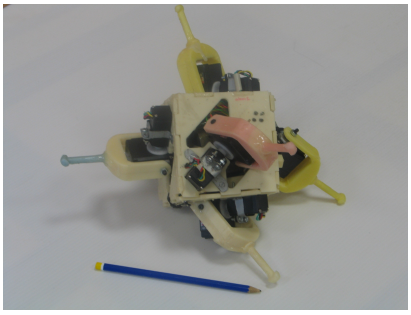
From the human point of view it is then difficult to have simultaneously an overall vision of the group on three levels: the social level where we look for the global behaviour of any robot, the team level where we focus on a specific group of robot and the individual robot level.
The definition of our general language HoRoCoL is driven by these three levels of team programming: Social, Group, Agent. Social and Agent programming are very classical, the original part of this work is on the Group programming where we introduce two original instructions : ParOfSeq/SeqOfPar and the where instruction.

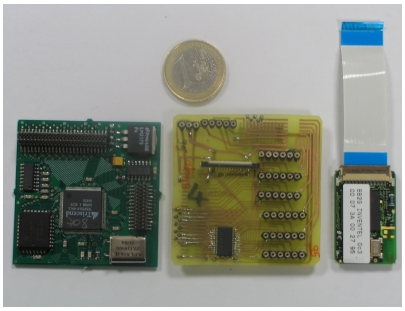
This paper presents the design of our robotics modular component, called Atom, and preliminary results on the prototypeand introduces the HoRoCoL language.

Hardware level

This section is a quick presentation of some mechanical aspects of the basic module (atom) and next, a description of the hardware and embedded software. Some informations on the progress of this project can be found in. The priority was given to the high-level tools and the communication middleware, .

Mechanical design
One atom is composed of six legs which are directed towards the six orthogonal directions of space. They allow the atom to move itself and/or dock to another one. The carcass of the atom consists of six plates molded out of polyurethane. A carcass weights approximately 180g. The first walking prototype of atom is shown here.





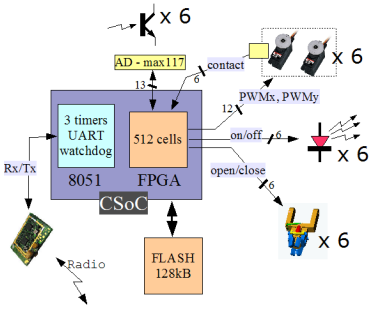


Fig1. -The fisrt prototype of Maam robot -The CPU Board -The CPU Organisation

The CPU has to
- control 12 axis (2 DOF for one leg) : each leg is driven by two servo--motors and a servo--motor is controlled by a PWM (Pulse Width Modulation) signal. The servo includes a motor, an angle reducer and a P.I.D. regulator.
- control the docking of two legs : the mechanic system under consideration provides a flip-flop control. The same control must alternatively couple then uncouple the two atoms.
- identify the legs at the touch of the ground : an atom may have 3 or 4 legs touching the ground at the same time. The presence of pincers at the tip of the leg make the installation of a sensor hard. We extract this information from the inside of the servo by processing some control-signals of the PID regulator.
- line up 2 legs : the mechanical connection between two atoms requires the lining up of two legs. We propose an infrared transmitter/receiver system. The search for an optimal position needs the use of 6 analog--to--digital converters for each atom. It may be useful to activate or deactivate the transmitter if necessary: that leads to add 6 digital outputs in our system.
- communicate with another atom or with a host computer: this aspect is discussed in the next section.

We also have the following general constraints for robotic and embedded systems:
- mechanical: the electronic is embedded in a robotic atom; it must fit in a cube which edges < 50 mm.
- adaptation: emergence of new requirements due to unforeseen problems during the development of robotic atom must not question the general architecture.

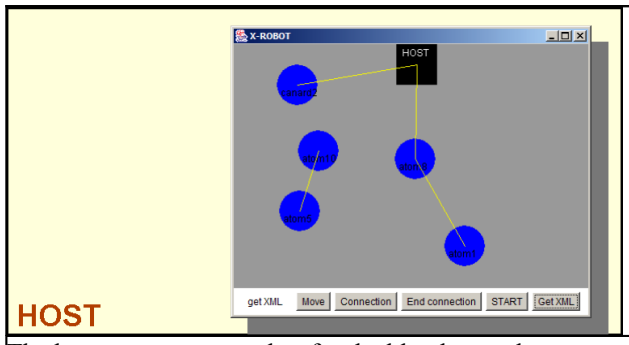
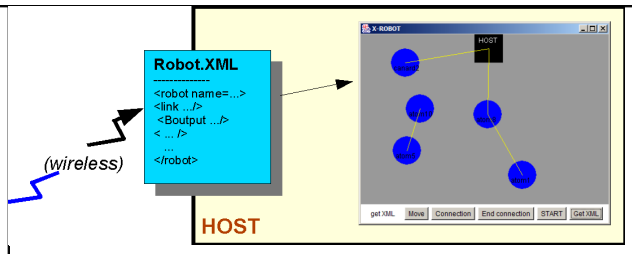
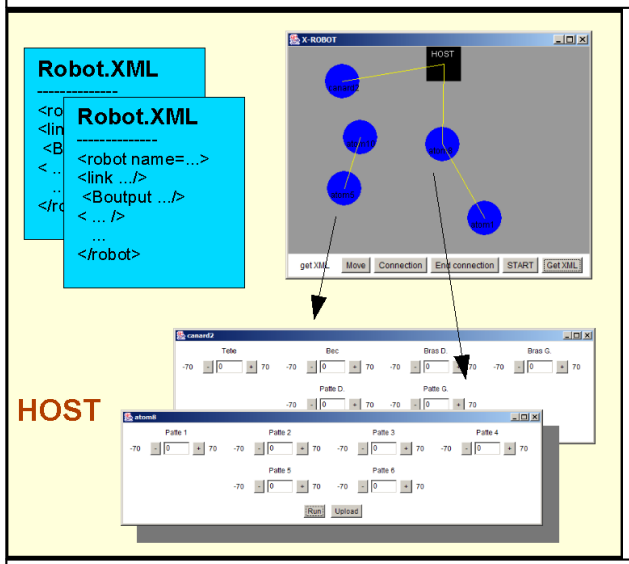
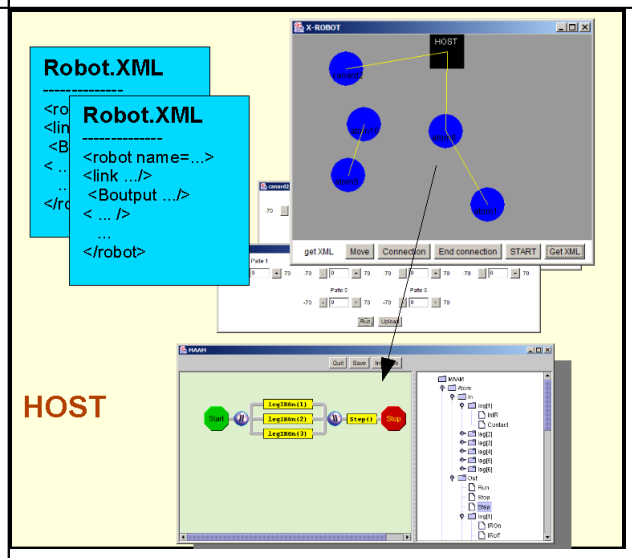
Embedded electronic

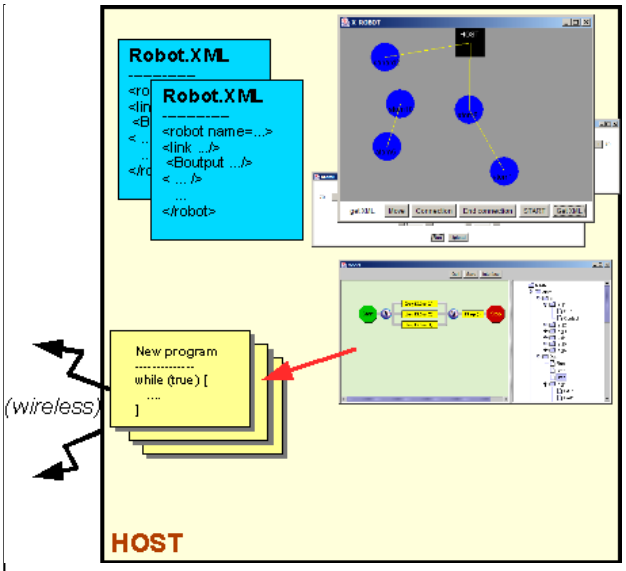
The architecture represented by the diagram in Fig 1 takes the previous enumeration of functions and constraints into account. The Embedded electronics is built around a Triscend TE505 CSoC. The TE505 integrates a CPU 8051, a FPGA with 512 cells and an internal 16KB RAM. It is completed by an AD convertor card and external bluetooth module for radio--communication

This solution gives a suitable answer for previous constraints. The micro-controller provides usual functions of a computing architecture: central unit, serial line, timers, internal memory. With the FPGA we can realise the equivalent of an input/output card with low level functionalities. It provide most of classical combinatory and sequential circuits (latches, counters, look--up--tables, comparators With the 512 cells build in the TE505 we could carry out the twelve PWM-commands, as well as the command of A/D converter (MAX117) in a pipeline mode and also other input/output. So we can command each axis by *just writing in one register* and control the level of the Ir receptor simply by reading in a register that is refreshed in real time.

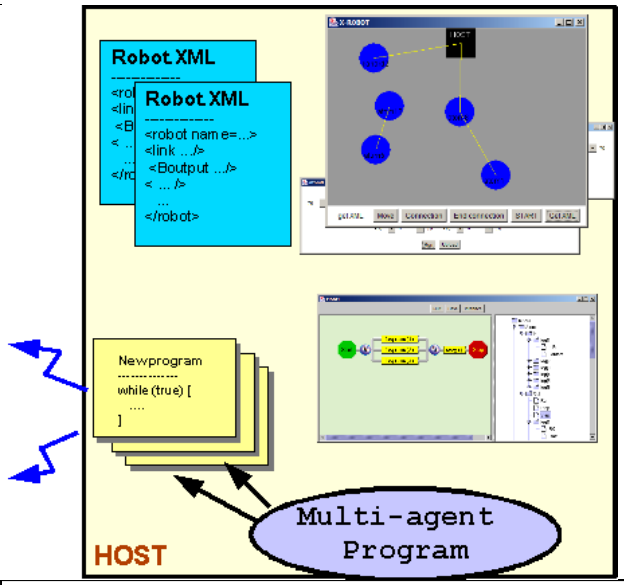
Low level Software

Because managing a team of robots with effectiveness suppose to guess the actual robots reachable in the area, to learn the capabilities of each one, to be able to distribute an application between them, and, possibly to remote-control any of them, we developed the following architecture.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>The host computer searches for the bluethoot robots accessible and build the first map of communication between them.</td>
<td>The host computer receives all the XML files coming from the robots. These files describes the commands and sensors available on the robot.</td>
</tr>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>A specific interface allows to make a direct use of the commands avaliable on each robot</td>
<td>It is also possible to use the generic interface to program the all set of robots using the Horocol language (decribed further).</td>
</tr>
</table>



After a compilation, the program can be loaded in each robot connected to the system. In each robot a **local language interpreter** will run its own part of the code.



This general procedure permit to program with a unify system a all set of robot making by this a multi-agent programming language : Horocol.

Fig 2 : Ambient Robotics presentation

Horocol Language

In Horocol we distingush three levels of programmind : social programming, coordination programming and agent programming

Social programming

Is used to express the general behaviour of the sets of agent. It gives a general description of « what set of agent is doing what ». It is a meta langage of behaviour. It offers a high level point of view to the programmer who describes its calculation in term of composition of event, directed parallel programs synchronized by areas. Areas stands for a set of agents, virtually or physically distributed over a network of computers or a set of mechatronics agents, running the same goal.

Main structure

```
Horocol : :=      *import file.xml ;*
                  programHorocol program_name {
                        agents_set_declaration          // Declaration section
                        * global_instruction ; *                // Programming section
                  }
```

import is used to express what which real robots are used in this program.

We assume that an agent (robot) is define by a set of 3 files:

- primitive.xml which describe the elementary actions that can perform the agent (for instance in the next example myRobot will be able to : “findTheBall() or searchNeighbourg() …”

- langage.xml which describe the kind of program that can execute the agent (robot). This language can be a standard programming language (Java, C++, …) or a specific language for an industrial robot. In our case it is the interpeter described in Fig2 which is the target language.
- horocolSystemBasics.xml which describe the list of system features available for this agent (robot) like communication, synchronisation... This file will allow a Horocol engine to know if it is possible to generate from an Horocol program a specific program for this agent (robot).

These 3 files are supposed merged in a file file.xml. In the case of a set of heterogeneous robot then there will be several different file.xml : one by kind of robot.

Declaration section

[A] agents_set_declaration : := *agents_type_declaration* [A1]
*agent_list * [A2]
[social_variable] [A3]
[social_event] [A4]

Agents, variables and events are declared at the global scope. Agents list is the list of all agents participating to the program. Variables and event declared at global scope are supposed visible by any agent. The public variables of the agents (define in files.xml) are also supposed visible by all agents.

[A1] agents_type_declaration ::= **type** agents_type_identifier **use** file.xml**;**

This construction defines type of agents and make the link with the real external agents/robots.

[A2] agent_list ::= agent_type_identifier identifier**=newAgent(**[agent_type_identifier]**);**

This is the declaration of all the agents participating in this code. The **newAgent** order express the begining of the robot life for this appication. *It is not necessarily implementable it can be reduce to "power on" a robot*.

[A3] social_variable ::= type_indication identifier_list [**limited**(agents_list, agents_type)] [**=** expression]**;**

Classical variable are allowed (int, float, boolean …). This defines public variables visible by all agents of the system named social variables.
The keyword **limited** express that for the agents in agents_list or agents_type the variable is "read only".

[A4] social_event ::= **event** identifier_list;

It is possible to declare public events. Social event are supposed to be visible from all agents declared in this program.

Programming section

[B] global_instruction ::= global_noninterrupt_action [B1]
| global_interrupt_action [B2]
| global_parallel [B3]
| global_variable_assignment [B4]
| global_if [B5]
| global_loop [B6]

[B1] global_noninterrupt_action ::= **[** local_program **]**

This defines a program that is executed from the first to the last instruction without possibility to end its execution. This construction is only usefull in [B3] construction.

[B2] global_interrupt_action ::= ° local_program °

This defines a program that can be ended during its execution. The way of finishing a program is not defined by Horocol because it depend on the type of real agents used. Basically we can consider two kinds of ending. First the system kills the program. The second way is to send a message to this program to ask it to finish, this technique will be preferred when security is needed (shared variable, robot …).

[B3] global_parallel ::= **||** **(***global_instruction**,*** global_instruction **)**

This construction allows to begin at the same time two (at least) different programs over the set of all agents. At this point an agent will execute the first code possible for him. This means that in the case of a **|| (**P1**,**P2,P3,P4**)** then there is 4 programs running in parallel. An agent will execute the first program that he is able to execute in the list beginning by P1 end ending by P4. The **||(**P1**,**P2,P3,P4**)** instruction is terminated when all the programs P1, P2, P3, P4 are terminated.

[B4] global_variable_assignment ::= identifier **=** expression

[B5] global_if ::= **if** (test) **{** local_program **} else {**local_program**}**

[B6] global_loop ::= **while** (test) **{** local_program **}**

These are very classical assignment to social variables or If and While instructions

.

Coordination or Group Programming

The coordination programming gives the description of how a specific set of agent will execute the code and how it is distributed over this set of agents. This is the most original part of the language. It contains two different original constructions.

First original part : the couple **seqofpar** and **parofseq** express the way in which the code is executed : synchronously (**seqofpar**) or independently in parallel (**parofseq**).

Second original part : the **where** instruction express the pre condition to be satisfied for an agent to execute a code.

```
[C]   local_program : :=
            *[global_variable_assignment]*          [B4]
          | *[agent programming]*                   [C1]

[C1]  agent programming :: =
            <agent name>.method()                   [D1]
          | seqofpar(agent_type_list) {             [LP1]
                [protected_declaration]             [D2]
                *where_without_event*                       [D3]
            }
          | parofseq(agent_type_list ){             [LP2]
                [protected _declaration]            [D2]
                *where_with_event*                  [D4]
            }
```

Coordination programming in Horocol language takes three different forms : a classical specific method called to a specific agent [D1], or one of the two constructions "**seqofpar**" [LP1] or "**parofseq**" [LP2] detailed below.

```
[D2]  protected _declaration ::=
          type_indication identifier_list [ limited( agents_list, agents_class)] [= expression] ;
        | event identifier_list ;
```

It is possible to declare protected scope variables or events (inside a **seqofpar** or **parofseq**). In this case they are visible only to the subset of all agents executing this part of code.

[LP1] Instruction **seqofpar**: sequence of parallel

seqofpar(agent_type_list**)** can be understood as : "apply seqofpar to all agents having the type « agent_type » in the following". In this construction « agent_type » are defined in **[A1].** The **seqofpar** is a control structure for which each line of the internal program (**where_without_event**) will be executed synchronously over all agents concerned by this branch. Synchronously execution means that agents execute one instruction at the same time than the others.

```
[D3]  where_without_event ::=
          where (test) {
              [private_declaration]              [D5]
              * local_instruction ; *           [E]
          }
```

where indicates who is concerned by the "local_instruction". This construction can be understood like : "for all agents satisfying the condition expressed by the **test** execute the following **local_instuction**". Remark also that it is possible to define private variables [D5] or events which are visible only by the agent executing this branch an duplicated in each of them (i.e. duplicated in each agent satisfying the condition **test**). Because this kind of instruction is in a seqofpar this means that each instruction [F1] to [F11] of the local instructions [E] is executed locally at the same time on each agent satisfying the **test**. In this case we speak of synchronous multi-agent programming.

[LP2] Instruction **parofseq**: concurrency of sequence

Here all agents concerned by the code (**where_with_event**) are executing their code in parallel and no instruction synchronisation between them is made. This means that all agents execute its own code independently from the others. The only instruction synchronisation is at the end of the **parofseq** because this instruction is considered terminated when all the agents concerned by the internal code have finished their execution.

```
[D4]  where_with_event ::=
          where (test ){
              [private_declaration]              [D5]
              * local_instruction ; *           [E]
          [react
              * when_event ; *]                 [D6]
          }
```

```
[D5]  [private_declaration]
```

Declaration of private variable or event at a private level. These elements are only visible by the agent executing this code and are duplicated in each agent.

[D6] when_event ::= **when** test **=>** * local_instruction **;** *

The **react** part is used to express reactive multi-agent programming. It works like exceptions in standard languages. Each time that an event (supposed visible by the agent executing this code) is emitted (by the **emit** [F7] instruction) then the **react** part is activated and looks if a specific program is linked to it. If it is the case then this program is executed else the normal program continues at the point where the event arrived.

If during the execution of local_instruction an other event is raised the this second event is queued until the end of the code actully running in the react part. After what it will be treated by the react part. *Nevertheless*, i*t is possible to use Horocol to simulate preemption and priority.*

Agent programming

The agent programming will describe the code executed at low level by the agents. To the set of instructions defined in [E] we have to add local agent primtives which are defined in the file.xml imported in the beginning of the Horocol program.

[E] local_instruction : :=

basic_primitive()	[F0]
<agent>. basic_primitive()	[F1]
| **if (**test**) {** local_instruction **}{** local_instruction **}**	[F2]
| **while (**test**) {** local_instruction **}**	[F3]
| **loop** local_instruction **end loop**	[F4]
| **exit**	[F5]
| variable_assignement	[F6]
| **emit** event	[F7]
| **resume**	[F8]
| **restart**	[F9]
| **reevaluate**	[F10]

[F0] basic_primitive()

Again classical specific method applied to the concern agent.

[F1] <agent>. basic_primitive()

Again classical specific method call to a specific agent identical to[D1].

[F2] **if (**test**) {** local_instruction **} else {** local_instruction **}**

standard.

[F3] **while (**test**) {** local_instruction **}**

standard

[F4] **loop** local_instruction **end loop**

standard

[F5] **exit**

used to exit from a loop …end loop [F4] or while [F3] instruction.

[F6] variable_assignement

Identical to [B4]

[F7] **emit** event | **emit** event (*type var,* type var)

This instruction emits an event that can be declared at the social level [A4] or protected if declared in [D2] or private if declared in [D5]. When an event is emitted then the react part [D6] of the program is executed.

[F8] **resume**

This instruction can be present only in **when_event** [D6] part. Its execution will restart the execution of the corresponding **local_instruction** [E] program at the instruction where was emmited the event that stops its execution to enter in the **react** part.

[F9] **restart**

This instruction can be present only in **when_event** [D6] part. Its execution will restart the execution of the corresponding **local_instruction** [E] program at the first instruction.

[F10] **reevaluate**

This instruction can be present only in **when_event** [D6] part. Its execution will restart the execution of the **seqofpar** or **parofseq** instruction. The idea is to check is the agent stills have the properties expressed in the **where test** of [D3] or [D4]

Example of Horocol programming

Let's consider an example in which we simulate the behaviour of a robot team playing football. Let's say that there is 4 players, one goal keeper and one coach in the team. A general clock will calculate the end of the play.

```
import myRobot.xml;
import clock.xml;
programHorocol footballVersion1
  type football use myRobot.xml;                    // the football type is builded by the decscription of my robots
  type clock use  clock.xml;                        // a general clock type
  football        a1,a2,a3,a4,a5,coach = newAgent( football ); // define 6 agent variables member of the team
  clock           watch= newAgent( clock);          // and on agent for the clock
  event           coachGivesOrder, timeOut;         // two global events one for the coach, one for the clock
  int             coachStrategy;                    // the global variable defining the strategy of the team
  int             time limited (football);    // variable impossible to modify for agent having type football
{
// init part here we suppose that each football agent will receive his assignment (player, goal keeper, coach)
 parofseq(football, clock){
        int playersOrganisation ;   // this variable is visible for all members of this section
        where (football.isPlayer()){        // only football agents satisfying isPlayer() primitive execute this
section
            football x;                     // this local variable is in each player of this section
                loop
                    findTheBall();     // call to a primitive of the agent defined in myRobot.xml
                    if (foundBall ) { moveToBall(); shootBall(); }
                    else { localMove(playersOrganisation);
                        x =searchNeighbourg();  // search an other agent to speak with
                        playersOrganisation = x.exchangeInformation(); // discuss with this agent
                    }                   // this make possible change in the team organisation
                end loop;
            react                   // if an event is raised the previous section stops and this part is executed
                when timeOut                => resetBehaviour() ; restart; // end of the game start again
                when coachGivesOrder        => setMyself( coachStrategy); reevaluate; // change my behaviour
        };
```

```
where (football.isGoalKeeper()){ // executted by the football agent verifying isGoalKeeper()
        loop
          findTheBall(); ….
          coach.exchangeInformation();// direct talk between the goal keeper/coach: local synchronisation
        end loop;
      react
```

```
            when timeOut                    => resetBehaviour() ; restart;
            when coachGivesOrder            => setMyself( coachStrategy); reevaluate;
};      // according to the coach decision the goal could change his behaviour and become a player
```

```
where (football.isCoach()){
   int coachConclusion ;            // local variable used by the coach to analyse the situation
        loop
          if (time<100) { coachStrategy := 15;  emit coachGivesOrder;} // ending game changes strategy
           else {coachConclusion = analyseSituation();
                if (coachConclusion != coachStrategy){   //comparison general strategy local conclusion
                  coachStrategy = coachConclusion;       // define the new strategy
                  emit coachGivesOrder;}                 // raise an event to "react" other player
                 }
        end loop;
};
```

```
where (clock.isWatch()){
     while (time>0) { waitOneSecond(); time =time-1;};
    emit timeOut;          // raise the timeout then all the agents will react to this event and end the game
};
```

```
}
}
```

Mapping Horocol on a set of real robots

By these example we see how the Horocol programs are linked to the real robot by the use is the **import** and **use** constructions.
The idea of Horocol is to assume that some primitive actions are available for each type of agents. Then when we write an Horocol program we manipulate these primitives under some parallel : ||, seqofpar or parofseq constructions. In fact, depending on the hardware structure of the robots, we have no guaranties that these parallel constructions are really possible to implement. For instance if the robots are very simple : contact sensor, ligth sensor no communication (think of a Lego Mindstorm robot) then constructions like : seqofpar or a direct call to a specific robot [F1] are not possible.

To know if it is possible to compile the Horocol program in a equivalent code running on the real robot the Horocol compiler will use the informations included in the XML file. This file is including three levels of information :
- the *robot primitive*,
- the *syntax of the language* used to program this robot,
- the *horocol system* primitives avaliable on this physical target.

The compiler checks first with the information stored in the *horocol system* if all the basics features exist to implement : social or protected variable, parallel constructions, direct information exchange.
The second phase is to check if all the primitive used in the Horocol for the associated type are present in the *robot primitive*.
Finnaly a purely syntactic rewriting transform the Horocol source code in the specific robot language. Of course this last pass is specific to each robot language so it needs to be rewrited for each kind of target. In our case we tested this transformation for the local interpreted language mentioned in fig 2.

CONCLUSION

The Horocol language proposed here allow the description of multi-agents, multi robots behavior at three different levels : social, coordination and agent. The originality of Horocol is in the instructions : **parofseq/seqofpar** for synchronous programming coupled to the **where** instruction for precondition evaluation coming with the **reevalute** to check for dynamical. Coupled to the distributed hardware it offers a solution to distributed mechatronics systems.

References

[1] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, S. Koraji, "M-TRAN: Self-Reconfigurable Modular Robotic System" in *IEEE/ASME transactions on mechatronics*, Vol.7 No.4 2002

[2] M. Rubenstein, K. Payne, W-M. Shen, "Docking among independent and autonomous CONRO self-reconfigurable robot" in *ICRA 2004*

[3] M.W. Jorgensen, E.H. Ostergaard, H. Hautop, "Modular ATRON: Modules for a self-reconfigurable robot" in proceedings of 2004 *IEE/RSJ International conference on Intelligent Robots an Systems* (IROS 2004).

[4] http://robotics.standford.edu/users/mark/polypod.html

[5] C. Unsal and P.K. Khosla, "A multi-layered planner for self-reconfiguration od a uniform group of I-cube modules", IEEE/RSJ, IROS conference, Maui, Hawaii, USA, pp 598-605, Oct. 2001. http://www-2.cs.cmu.edu/~unsal/research/ices/cubes

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

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

[8] P. Hudak & all "Arrows, robots, and functional reactive programming" *LNCS 159-187* Spinger Verlag 2002

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

[10] F. F. Ingrand & all "PRS: a high level supervision and control language for autonomous mobile robots", *IEEE Int Cong on Robotics and Automation* Minneapolis, 1996

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

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

[13] C. Zielinski "Programming and control of multi-robot systems" Conf. On Control and Automation Robotics and Vision *ICRARCV'2000* dec 5-8 2000, Singapore

[14] J. Armstrong "The development in Erlang", *ACM sighpla international Conference on Functional Programming* p 196-203. 1997

[15] M.S. Atkin & all "HAC: a unified view of reactive and deliberative activity". Notes of the *European Conf on Artificial Intelligence 1999*

[16] G. King "Tapir: the Evolution of an Agent Control Language" *American Association of Artificial Intelligence* 2002.

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

[18] J. Peterson & all "A language for declarative robotic programming" *Int Conf on Robotics and Automation ICRA 1999*

[19] E. Klavins "A formal model of a multi-robot control and communication task" *IEEE Conf on Decision and Control*, 2003

[20] E. Klavins "A language for modeling and programming cooperative control systems" *Int Conf on Robotics and Automation ICRA 2004*

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

[22] F. Mondada & all "Swarm–bot: for concept to implementation", IEEE/RSJ Int Conf on Intelligent Robots and Systems IROS 2003