Using HoRoCoL to Control Robotics Atoms

Y. Le Guyadec, C. Guégano, M. Dubois, D. Duhaut
Valoria
University of South-Brittany
Vannes – France
Contact: 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 and two possible execution models. The expressivity of the language is illustrated on a simple example. Specific compilation problems are discussed.**

I. 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]. These robots generally consist in modules working together and where each module is permanently linked to at least one other.

Some inspiration of our robot comes from the biological world of the ants: each one has a certain autonomy, but they can help each other to achieve particular tasks (e.g. building a bridge). It consists in several autonomous entities, called Atoms, due to their physical look (see Fig. 1).

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 (section II) of our robotics modular component, called Atom, and preliminary results on the prototype. Section III introduces the HoRoCoL language. Its expressivity is illustrated on a simple example (section IV). Section V summarise specific compilation problems.

II. Hardware architecture of MAAM

In this section, we summarize the main aspects of the MAAM hardware architecture which is the experimental physical platform for HoRoCoL. We will first present the mechanichal and electronic aspects, and after, we will briefly see the communication system.

A. Electronic and mechanical features

An atom is composed of six legs which are directed towards the six orthogonal directions of space. They allow the atom move itself and/or couple to another one. The first walking prototype of atom appears on Fig. 1. This prototype embeds all the electronic and software functions described in this paper. It does not include the pincers.

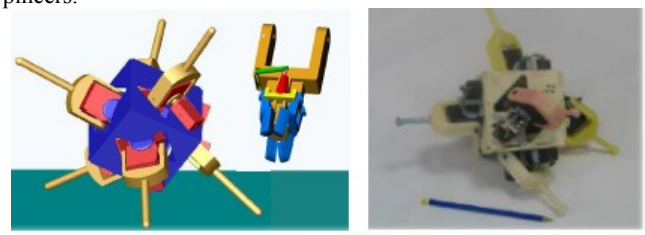

Figure 1. Two steps in the project: simulation and walking prototype.

The control/command system has to :

(i) Control 12 axis (2 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.

(ii) Control the coupling of two legs: the mechanic system under consideration provides a flip-flop control. The same control must alternatively couple then uncouple the two atoms.

(iii) 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 pincers make the installation of a sensor hard. In our case, this information is obtained by processing some control-signals of the PID regulator.

(iv) 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 research for an optimal position needs the use of 6 analog-to-digital converters for each atom. It may

be useful to activate or desactivate the transmitter if necessary: that leads to add 6 digital outputs in our system.
(v) Communicate with another atom or with a host computer: this aspect is discussed later.

The architecture represented by the diagram in Fig. 2 takes the previous enumeration of functions and constraints into account.

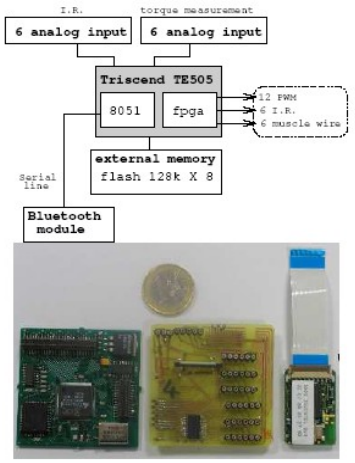

Figure 2. Embedded electronics : TE505 CSoC with external memory, AD convertor card and external bluetooth module for radio-communication

It is build around a configurable system on chip (CSoC), which integrates a micro-controller and a FPGA (Field Programmable Gate Arrays) in a single component. The micro-controller provides usual functions of a computing architecture: central unit, serial line, timers, internal memory,... FPGA enables to realize the equivalent of an input/output card with low level functionalities. It provides most of classical combinatory and sequential circuits (latches, counters, look--up--tables, comparators ...). We've opted for the Triscend TE505 CSoC. This component integrates a CPU 8051, a FPGA with 512 cells and an internal 16KB RAM.
Inputs an outputs: all IO functions are distributed among FPGA and external cards. As many as possible functions are embedded.
PWM control: position control of servo-motors is obtained PWM. The position is proportionate to the width of a periodic pulse. The period is about 20 ms, the range of the pulse width is from 0.9ms to 2.1ms The servo performs the position loop.and the FPGA provides the 12 PWM control signals. **Contact detection** is taken from the internal circuitry of the servo-motor. By processing the command pulses with some combinatory logic and a retriggerable monostable we get a boolean information.
Infrared positioning: coupling one atom to another involves lining up two legs. The accuracy of alignment will be fixed by mechanic constraints of the system under development. The solution which has been carried for lining up two legs implements infrared transmitter/receiver. It involves a dialog between two atoms that become alternately transmitter then receiver. To do that, 6 channels digital-to-analog converters are required, and 6 outputs (taken on the CSoC) to control the transmitters. Analogic to digital conversion is done with a max117 working in pipelined mode. Signals used for driving the convertor in this mode are also generated by the FPGA, so we have just to read data registers.
Coupling control: for coupling one atom to another, we have to activate a pincer. Due to size and weight feature, we propose to drive the pincer with a SMA affector (shape memory alloy).

B. Embedded software
The dynamic aspect of the embedded software is described in Fig. 3. The interpreter is the normal background task. Communication and program management (upload and download) are coded in interrupt routines. When the connection is established, a new program can be uploaded on the fly and launched immediately. In a centralized approach, the main program is desactivated, and robots execute simple orders received from the communication layer.
An important point of view in our approach, is the modelisation of low level functions of the robot in a XML file. The robot keeps in memory a card of all its capabilities, and can export it towards a host. With this feature, a priori knowledge on robot capabilities is not required, provided that it gives its own interface (DTD).
The interpreted program uses only the instructions described in the interface, and, in a centralized control, the HoRoCoL program sends orders according to this interface.

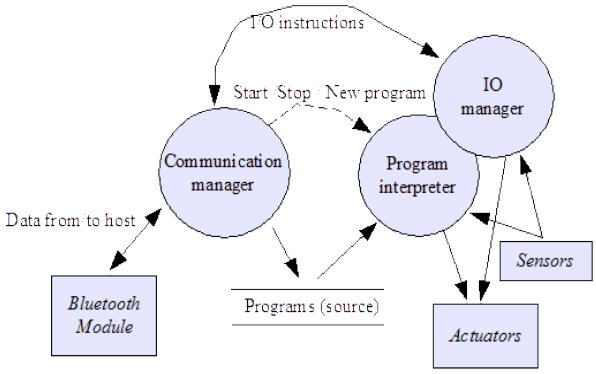

Figure 3. General architecture of the embedded software. The communication manager receives orders and new programs. It also manages bluetooth connections and disconnections. The program interpreter is activated or stopped by the communication manager.

C. Communication between modules
The communication is an inescapable aspect in a multi-agent architecture. The characteristics of the selected technology has consequences on the final software architecture. An atom must communicate with one or more of its neighbors (autonomous approach), and/or with a host-system (centralized approach). Our realization is build with bluetooth technology that gives us suitable responses for noise constraint, miniaturization of modules and low cost. We use it just above the HCI (Host Control Interface) layer. That allows us to control the complete bluetooth device (link management and baseband). HCI packets contain either user data or command. All layers from radio to HCI are implemented in the industrial module (we can see it on Fig. 2).

There are several ways to manage a set of communicating robots with bluetooth: we can let the modules inquire after the host and require a connection link when they found it; we can broadcast messages for all robots in the bluetooth area; we can try to establish dedicaced links with all the detected modules. In a first approach, we have chosen the third solution. We can create links between the host and robots, and ask a robot to connect to another one to transmit received orders to it. This leads to an ad-hoc network. The communication between agents is held by a class `SetRobots` wich provides services (connection, disconnection, inquiry ...), and notifies all suscribed tasks for events. This class is reusable for many applications. Fig. 4 shows a set of 8 modules detected in the bluetooth area.

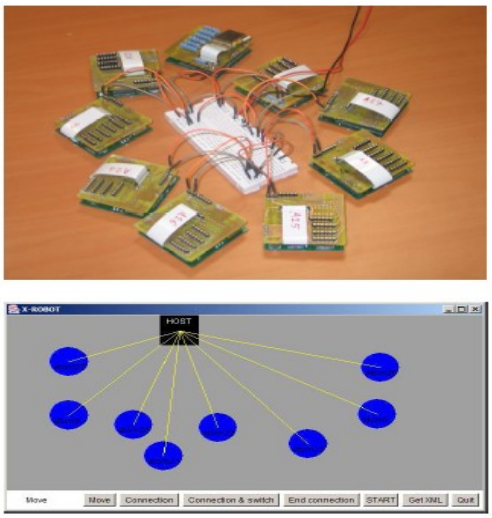

Figure 4. Eight CPUs are communicating with the host. Basic commands, and/or request for XML description can be performed.

III. The HoRoCoL language

HoRoCoL stands for Homogeneous Robotics Component Language. It was initialy designed to control specific agents, but its features also allow to control generic ones. It enables to describe the behaviour of sets of homogeneous agents (deliberative part plus actuative part) in a unique centralized program (expressed in an extended Java like syntax) which communicates and synchronizes local or distant agents. Thus, it takes advantage of the parallel model with shared variables for high level control (global shared variables are used for communication purpose), but also offers the parallel model with distributed memory (where communication are implemented via message passing). Furthermore, following the SPMD programming model (Single Program Multiple Data) [24], HoRoCoL implicitly works on the set of all agents: it is not necessary to name agents for methods invocation. This property facilitates scalability, test and reuse. The main problem with writting classical parallel programs is to ensure consistency of shared variables and absence of deadlock. Thus, HoRoCoL offers implicit synchronization mechanisms and guaranties consistency of access.

Supported agents are implemented via active objects (software agent) or reactive systems (robotic agent) doted with communication features (bluetooth in our experiments). From the programmer's point of view, each agent embeds a XML interface it exports to a dedicaded host. This interface is imported into a HoRoCoL program using the `import` keyword. It presents the list of functions implemented by an agent. The DTD (Document Type Definition) attached to each agent type, associates to each operation an information on the communication protocol and gives an objet oriented approach to the description of an agent type (especially in the case of robotics agents). The HoRoCoL language is thus independend of communication protocols and synchronisation mechanisms. It uses this interface has an entry point to communicate with agents.

In HoRoCoL, a high-level point of view is offered to the programmer who describes (deliberative) control of agents, according to 3 layers in the language : social, group and local. Local programming can be expressed using any language supported by the agent, provided that its XML interface is exported to the dedicated host. The chalenge of compiling such programs is to feel the gap between this centralized point of view and distributed asynchronous executions.

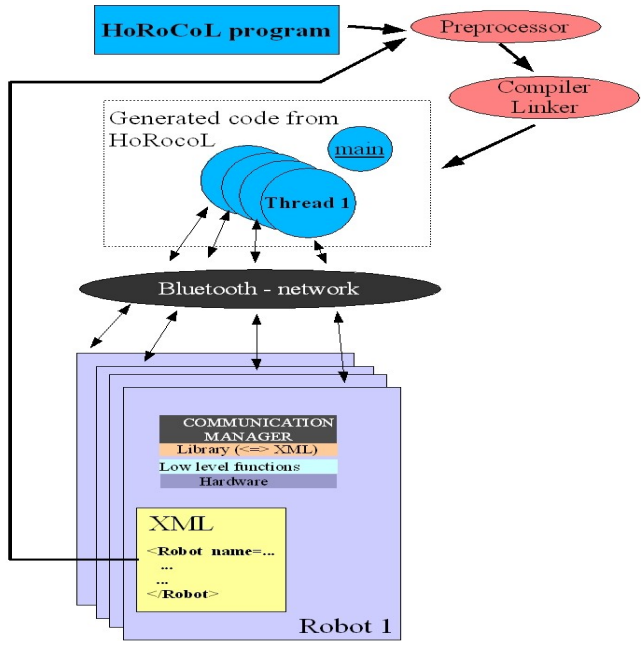

Figure 5. General Architecture – centralized model

Two execution models are investigated: centralized and distributed. In the centralized execution model (described below), a dedicated server loads and interpretes the HoRoCoL program, runs it localy communicating with distributed agents. This execution model is very close to the abstract point of view proposed by HoRoCoL. In the distributed execution model a local version of the HoRoCoL program is replicated onto each agent. Many problems have to be adressed in this case: they are discussed in section V (*Implementation and Compilation Issues*).

In this paper, we only focus on the definition of the centralized execution model (see Fig. 5).

A. Social layer

This layer allows to control (possibly empty) sets of agents - for instance *maam*, *players* or *clock* – in a parallel shared point of view (shared scalar variables, shared events, seq, if, loop, parallel). The following skeleton shows a typical social layer program:

```
import agentType1.xml;
import agentType2.xml;
program_Horocol aSqueleton {
  agentType1 use agentType1.xml;
  agentType2 use agentType2.xml;
  event aGlobalSharedEvent;
  int aGlobalSharedInt;
  agentType1 a1,a2 = newAgent(agentType1);
  agentType2 a3,a4 = newAgent(agentType2);
  ||(°GP1°,[GP2]);
}
```

In this example, two kind of agents are manipulated. Their functionnal interface is imported and 2 first-class type identifiers are declared: `agentType1`, `agentType2`. Agents of that type are allocated, then programs `GP1` and `GP2` runs in parallel on distinct sets of agents, depending of agent type conditions (see section group layer below). The construct `||(GP1,GP2)` enables parallelism. It terminates when `GP1` and `GP2` have terminated. The skeleton also illustrates interruptible program `°GP1°` and

uninterruptible program [GP2]. The informal meanning is that °GP1° terminates as soon as GP2 does.

Communications are possible using shared variables. Their implementation has to ensure consistency of access. They are manipulated using classical sequential control structures. This sequential flow control can spawn into several flow control (threads). Each (distant) agent is then represented by a centralized thread, called *proxy thread*, running its private version of the HoRoCoL program. This thread chooses its branch of social parallelism according to agent-type conditions, declaring/accessing local variables.

B. The group layer

This layer allows to describe asynchrony (ParOfSeq), synchrony (SeqOfPar) and events. The ParOfSeq/where construct allows to describe asynchronous concurrency in a single program manner. We consider the following abstract skeleton of the ParOfSeq construct:

```
ParOfSeq(agentType) {
  event aSharedEvent;
  int aSharedInt;
  where (localCondition1) {
    event aLocalEvent;
    int aLocalInt;
    P1;
  }
  where (localCondition2) {
    P2;
  }
react
  when aSharedEvent => P3;
  when aLocalEvent  => P4;
}
```

In this program, *agentType* enables to select a type of agents. For instance *players* or *maam.* The following bloc only applies to that subset of (homogeneous) agents. Each proxy thread agent chooses a where branch regarding to local conditions stored in the agent (to which one can access using its published XML interface). The associated bloc (P1 or P2) is then executed asynchronously. In this bloc, local or global events can be emitted using the emit construct. Local events are declared and allocated by any proxy thread running the considered bloc: they can be used to program reactive behavior of a given agent. Shared events are broadcasted to all active agents in the current scope. Events are treated in the react bloc associated to each ParOfSeq. Specific keywords are available here: resume, restart and reevaluate.

- resume enables to continue the current where in sequence.
- restart enables to reexecute the current where bloc, restarting at its first intruction.
- reevaluate enables to reexecute the current ParOfSeq bloc. So a new where branch can be choosen, depending on the (changed) internal state of the agent.

Note that the react bloc and the associated keywords resume, restart and reevaluate are only available in the ParOfSeq construct. This is due to the asynchronous underlying paradigm which associates a proxy thread to each agent. In a synchronous bloc, this arise to ambiguous semantics.

In the SeqOfPar/where construct, agents execute their where bloc synchronously. Two cases can be studied, depending on the number of where branches attached to the current SeqOfPar (2 branches in the skeleton below).

```
SeqOfPar(agentType) {
  int aSharedInt;
  where (localCondition1) {
    int aLocalInt;
    I11;
    I12;
    I13;
  }
  where (localCondition2) {
    I21;
    I22;
    // skip inserted
  }
}
```

In the case of a unique where branch, agents that satisfy the where condition execute the associated bloc synchronously. The complementary set is desactivated for the rest of the current SeqOfPar, waiting for the active set to terminate its where bloc. Note that since HoRoCoL offers a high level point of view, the grain of synchrony is the instruction (HoRoCoL instruction or message sent to a distant agent). So, for a given set of agent, all occurs as if a synchronisation barrier was placed at the end of each instruction. In the case of several where branches in the same SeqOfPar, each proxy agent localy decides which where branch (possibly none) it will execute depending on the evaluation of local conditions. Then, all the branches are computed synchronously: the second instruction of each bloc (I12 and I22) starts when the first instruction of ***all*** the where branches (I11 and I21) is terminated for all active agent. In the case of dissymetric where branches (with different sizes), all occurs as if skip instructions (which does nothing and just terminates) where added at the end of each bloc to reach the right size (the size of the longest where bloc). In the case of loops, unrolled loops are considered.

IV. Using HoRoCol to control robotics Atoms: an example

In this section, the HoRoCoL architecture descibed below is applied to robotics Atoms. Fig. 6 shows the UML class diagram corresponding to the XML DTD exported by any Atom involved in a HoRoCol execution.

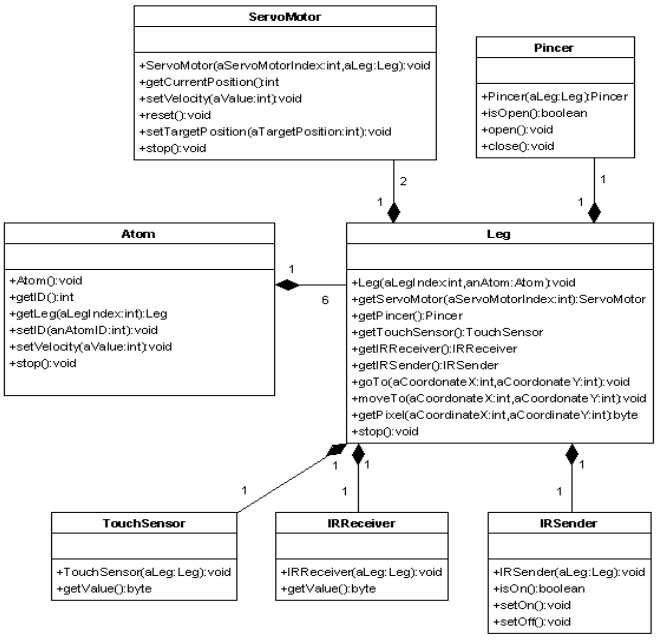

Figure 6. (Simplified) Class Diagram of an Atom.

Following the centralized execution model, it enables to communicates with distant atoms using classical remote method invocation. This class diagram[1] reflects the hierarchy of the mechatronics and mechanics components of an Atom :

- An `Atom` is composed with 6 legs and embeds several sensors.
- Each `Leg` offers 2 degrees of freedom driven by 2 `ServoMotor` that may be controled using position parameters : see the `get/setTargetPosition`, `goTo` and `moveTo` methods. `moveTo` offers control of trajectories (a straight line from origin to destination) while `goTo` does with the current servomotor speed (possibly segmented trajectories)
- Each `Leg` is equiped with an infrared transmitter/receiver system which is used either to line up 2 legs (ie computation of optimal positions) before mechanical connection arise, or to communicate between neighbors. These functionalities are available using `getValue/setOn/setOff`.
- Each `Leg` is also equiped with a `TouchSensor` that enables to deduce collisions, contact with ground, ... It should permit to infer knowledge on the current orientation of a given atom.
- A `Pincer` enables to connect Atoms.

This API is imported in the HoRoCoL example program below using the `MAAM.xml` file. The goal for a row of interconnected atoms is to walk during 3 minutes. In order to maintain it in balance, it is necessary to lift only one leg out of 2 per atom at a given moment.

Programming the social level consists in the instanciation of all agents, their initialization and the execution in parallel of two programs concerning each type of agent.

We make the assumption that the initial state (S_0) is a row of inter-connected atoms having their servo-motors in their canonical position i.e. legs of all the atoms are centered (see Fig. 7.a). To facilitate the comprehension of the source code, some constants are provided in the environment: `left` and `right` respectively indicate the index of the left legs and right ones which are on the ground. In the same way constants `min`, `middle` and `max` respectively indicate the minimal, default and maximum position of any servo-motor.

```
import MAAM.xml;
import Clock.xml;
programHorocol walkingRowWithTimeOut {
  type maam use MAAM.xml;
  type clock use Clock.xml;
  maam a1, a2, a3 = newAgent(maam);
  clock watch=newAgent(clock);
  // Suppose S0 be the initial state
  // Runs in parallel 2 control programs
  ||(°aRow°,[aClock]);
}
```

Program `aClock` runs a countdown and terminates after 3 minutes. The `Clock.xml` file defines the unique method `waitSec()`. After 3 minutes, the program `aClock` terminates, which causes the termination of the program `aRow` which was launched in interruptible mode as defined by the `°P°` construct.

```
int time=180;
ParOfSeq(clock) {
  where (true) {
    while (time>0) {
      // send message to wait one second
      waitSec(1);
      time--;
    }
  }
}
```

1 a full version is available at : `www-valoria.univ-ubs.fr/Dominique.Duhaut/maam`

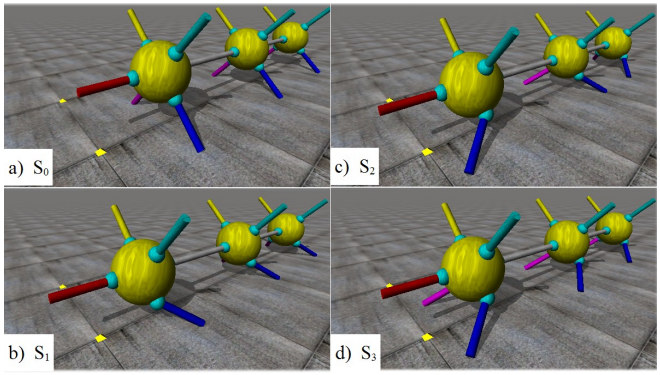

Figure 7. States S_0, S_1, S_2, S_3 of the main loop

Program `aRrow` describes calculation allowing the row to advance (1 iteration = 1 step for all the atoms). It is splitted in 3 phases which are repeated in a while loop (loop invariant: *all the legs on the ground are behind – see state* S_1).

- The first phase (from S_0 to S_1 or from S_3 to S_1, see Fig. 7.b) consists in positioning `left` and `right` legs behind. For that, it is not necessary to lift the legs. Since no synchronism between atoms is required, the `ParOfSeq` construct is used.
- In the second phase (from S_1 to S_2, see Fig. 7.c), left legs are moved ahead for the odd atoms and right legs are moved ahead for the even ones.
- The last phase (from S_2 to S_3, see Fig. 7.d) puts the complementary legs ahead.

These two last phases require a synchronism between atoms. Thus, there is a sequence of 2 `SeqOfPar`. In order to maintain the row in balance, we have to ensure that all the atoms have terminated the current `moveTo()` order of the `left` or `right` legs before starting the next move. When the goal is reached (the 3 minutes are spent), the program leaves the infinite `while` loop.

```
while (true) {
  ParOfSeq(maam) {
    // phase 1
    where (true) {
      getLeg(left).moveTo(min,min);
      getLeg(right).moveTo(min,min);
    }
  };
  SeqOfPar(maam){
    // phase 2
    where (getID()%2==1) {
      getLeg(left).moveTo(min,middle);
      getLeg(left).moveTo(max,middle);
      getLeg(left).moveTo(max,min);
    };
    where (getID()%2==0) {
      getLeg(right).moveTo(min,middle);
      getLeg(right).moveTo(max,middle);
      getLeg(right).moveTo(max,min);
    };
  };
  SeqOfPar(maam){
    // phase 3
    where (getID()%2==1) {
      getLeg(right).moveTo(min,middle);
      getLeg(right).moveTo(max,middle);
      getLeg(right).moveTo(max,min);
    };
    where (getID()%2==0) {
      getLeg(left).moveTo(min,middle);
      getLeg(left).moveTo(max,middle);
      getLeg(left).moveTo(max,min);
```

```
        };
      }
    }
```

As you can see, this algorithm is adapted to rows with unspecified size (scalability). Furthermore, its structure is close to intuitive mind.

V. Implementation and compilation issues

In this section, we discuss compilation problem and implementation issues, depending on the target robot (simulation under ODE[2] [25] or real Atoms) and the execution model (centralized or distributed). To achieve these different plateforms, several `MAAM.XML` files are under definition. They include information on :

- the *robot primitives* or local public variables specification;
- the *language syntax* used to program this robot;
- the *HoRoCoL system* primitives available on the target.

One can distinguish 2 kind of problems :

– Programming the robot primitives and building the corresponding `MAAM.XML` file depends on the robot underlying system. In case of a reactive system (our simulation approach), non-blocking messages are sent. So a mechanism of acknowledgment or future may be used to provide information on methods termination. This may be usefull to provide accuracy of movements. This is necessary to ensure the semantics of some HoRoCoL constructs : `SeqOfPar`, `ParOfSeq`, `||`. These mecanisms enable to build the necessary synchronisation barriers.

– The implementation of some HoRoCoL concepts (`SeqOfPar`, broadcast of events, shared variables, `react`, `resume`, `reevaluate`, `restart`) depends on the target execution model (centralized or distributed). In case of a distributed execution model, the HoRoCoL program is replicated onto each Atom. events, access to (virtualy) shared variables (physically replicated) and synchronisation mechanisms have to be implemented on top of the bluetooth network. Consistency of access to shared variables has to be ensured in this case. We study the possibility to use the Distributed Reactive Object Model [23] on top of the bluetooth device to have broadcast of events (in the centralized execution model, events are simply emitted and handled on the server side), the synchronization barriers and the global instant concept (shared logical clock).

VII. Conclusion

We have presented a high level set of tools for designing multi-robots architectures, and more particularly the Maam self-reconfigrable robot that is our contribution. Because the problems of motion for a crystal and docking between atoms are very hard, we propose to focus first on high-level tool build above a robust communication layer instantiated in the control system and in each module. Moreover, by using XML for robotic design, we get a methodological approach that should really improve the efficiency when designing new robots: the high-level tools don't change, and a great part of the embedded software remains the same.

HoRoCoL is designed on top of these developpements. It offers the programmer a high-level point of view by mixing different paradigms: parallel with shared variables, distributed with communication by message, SPMD. Specific constructs are proposed to deal with implicit synchronisations: `SeqOfPar`/`ParOfSeq`/`where`. We think this language is well adapted to modular robotics, but it can also be used to control software agents.

2 Open Dynamics Engine `www.ode.org`

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

[23] F. Boussinot, G. Doumenc and J.B. Stefani, Reactive Objects, INRIA Research Report RR-2664, Oct. 1995.

[24] Luc Bougé, The Data-Parallel Programming Model: a Semantic Perspective, INRIA Research Report RR-3044, Nov. 1996.

[25] M. Dubois, Y. Le Guyadec and D. Duhaut, "Control of Interconnected Homogeneous Atoms: Language and Simulator". Proc. of the 6th Int. Conf. on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR), pp 391-398