Proceedings 2005 IEEE International Symposium on
Computational Intelligence in Robotics and Automation
June 27-30, 2005, Espoo, Finland

Using Horocol to program a society of agents or teams of robots

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

Abstract **– In this paper we present an example of the use of the Horocol language for programming a society or teams of robots. This 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.**

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

I. Introduction

Expressing the code of the behaviour of teams of robots is a difficult task. It is due to the fact that robots are often physically programmed in different languages with very specific primitives linked to the hardware of the robot. We met this problem in two types of distinct applications: RoboCup [23] and the self reconfigurable robots [6, 7, 20]. In Robocup the teams of robot play football [19]. Robot are not industrial type. So, because they are "hand made", their programming is very different from one to another. It is thus necessary to be able to express when and how a robot player, which plays attacker, decides to become defender (and vice versa). Here we look for the reevaluation of a behaviour.

For self reconfigurable robots another problem is that, for instance, the walking motion implies synchronisation in the movement of the robot components. Then we need to express that all the robot modules participating in the movement, carries out their actions at the same time, synchronously.

We then notice that traditional languages we could use did not provide us simply these two constructions which appear useful to us: the expression of an attempt of reevalutation of its behaviour, the nature of the parallel execution of the group of robots.

In this paper in part II we make a quick review of current solutions for robot programming. In section III we give quick definition if the language Horocol, in part IV we present an complete example to show how is written the social programming and, at the team level how we can develop cooperation and coordination. In part V, we discuss the implementation of Horocol on a specific hardware system. Finally in section VI we give some conclusions.

II. Robot programming

Robot programming is a difficult spot studied since many years [13]. Often this field covers some very different concepts like : methods or algorithms (planning, trajectory generation...), or classically, architectures for robot control, usually hierarchical : centralised [1], reactive [8], hybrid [2, 9, 16]. Compared with these high level considerations, languages are developed in order to implement them [17, 21]. Different approaches appeared through functional [3, 4, 12] deliberative or declarative [5, 16, 18], synchronous [17] characteristics. In any way, we can schematically summarise the difficulties of robot programming in two great characteristics:

- one is that programming of elementary action (primitive) on a robot is often (even always) a program including many process running in parallel with real-time constraints and local synchronisation
- the other is that in its interaction with the environment the program driving the robot (sequence of primitives) must be able to carry out traditional features: interruption 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 which is that the programming is not reduced any more to a single physical system. The problem is then to program the behaviour of a group of robots or even a society of robots [10, 11, 14, 15]. In this case (except in the case of a centralised control) programming implies the loading in each robot of a program which is not necessarily identical to others because of the characteristic of the robots : different hardware, different behaviours and different programming languages. These various codes must in general synchronised to carry out missions of group (foraging, displacement in patrol...) and to have capacities of reconfiguration 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 the all set of robots, the team level where we focus on a specific group of robot and the individual robot.

0-7803-9355-4 / 05 / $20.00 ©2005 IEEE.

The main idea of our work is thus the give a formal definition of a general language, Horocol to express these three levels of team programming: Society, Group, Agent. Society 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.

III. Horocol introduction

The syntax of the Horocol language is described using a simple variant of Backus-Naur-Form. [item] express that the item is optional and *item* that item can be repeated. Keywords are in bold character.

A. *Social programming*

This section introduces the highest level programming in Horocol. The goal here is to express how the society of robot is composed and show its evolution. In this section we find classical constructions that can be compared to other robots or agent programming language see [17].

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

The use of **import** is discussed in the section IV. Let's assume that the file.xml gives the basic primitives of the robot.

Declaration section

This section is used to declare all the agents in the society and the list of variables, events used.

```
agents_set_declaration : :=
   *agents_type_declaration*
   *agent_list *
   *[social_variable]*
   *[social_event]*

agents_type_declaration ::=
      type agents_type_identifier use file.xml;

agent_list ::= agent_type_identifier
   identifier=newAgent([agent_type_identifier]);
```

This part define the set of types of robots and what external files give the "physical" implementation of these types. The list of agent instantiating the robots.

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

social_event ::= event identifier_list;
```

Define a list a social variables (with classical types : char, int, array ...) and event 'used for synchronisation). When defined at this level they are supposed visible by all agents. **limited** means that the variable is read only for a group. This definition does not suppose that the implementation over the real set of robot (xml files) is possible, see section IV on this subject.

Programming section

This section is used to express the general behaviour of the society of agents. Assignment [B4], condition [B5] and loops [B6] are very classical instructions.

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

[B1] global_parallel ::=
   || (*global_instruction,* global_instruction )
```

The [B1] expresses the concurrency between several programs. This means that all branch begin at the same time. It can be compared to *spawn* [3] or *compose* [11] or $P|Q$ [18] or even more classical *for each agent in* ... [12]

```
[B2] global_noninterrupt_action ::=
   [ local_program ]

[B3] global_interrupt_action ::=
   ° local_program °
```

[*P1*] defines a program *P1* that cannot be interrupted versus ° *P1*° defines a background task that will be interrupted when all branch are closed. For instance || ([*P1*],°*P2*°) expresses that *P1* and *P2* begins together but when *P1* will finish then it will stop *P2*. In the case of *P2* finishing first the || instruction waits for the end of *P1* to go in sequence.

```
[B4] global_variable_assignment ::=
   identifier = expression
[B5] global_if ::=
   if (test) { local_program } else {local_program}
[B6] global_loop ::=
   while (test) { local_program }
```

B. *Coordination programming*

This section is the most original part of this work. It is used to express how a team of robot is working together and how and when it reconfigure its behaviour.

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


[C1] agent programming :: =
```

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

A local program expresses how a group of robots is working. It can be restricted to a specific [D1] call to a primitive of an agent or use the seqofpar/parofseq construction.

seqofpar and **parofseq** is the second level of parallelism (the fisrts one is at the social level with || instruction). Here we are expressing the parallelism at the team level.

seqofpar is an control instruction for which each line of the internal program (*where_without_event*) will be executed synchronously over all agents concerned by this branch. This means that all the agents execute at the same time one instruction and move to the next one. On an other hand, the **parofseq** is an control instruction for which all agents are running their program in parallel with no synchronization.

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

Variables or events define at this level are visible only by the agent involved in this part of code (ie selected by the **seqofpar** or **parofseq** instruction).

```
[D3] where_without_event ::=
    where (test) {
       [private_declaration]          [D5]
       * local_instruction ; *        [E]
    }
[D4] where_with_event ::=
    where (test) {
       [private_declaration]
       * local_instruction ; *
    [react
       * when_event ; *]              [D6]
    }
```

The **where** instruction is used to select in a team of agents those having a true precondition. Notice that there can be several **where** in a **parofseq** or **seqofpar**. Each agent concern by this part of code (in the *agent_type_list* of **parofseq** or **seqofpar**) check the *test* of the **where** (from the first one to the last one). If the *test* is true then the agent will execute the internal code [D5, E or D6] else it will check the next **where**.

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

Variables or events declare at that level are only visible by the agent satisfying the *test* of the **where** instruction. These variables or events are duplicated in each agent.

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

In the case of a **parofseq** instruction a react part can be present. It looks like a exception treatment in standard languages. When an event is emitted (see [F7] in the next section) then the execution of the code is stopped and the control jump to the react part to look for a treatment to this event.

C. *Agent programming*

This part is used to define the behavior of a single agent. This isfinally what is "really" executed by the robot the rest of the code expresses when it begins, what code is executed, what variables or events are shared.

```
[E]    local_instruction : :=
   basic_primitive()                 [F0]
   |<agent>.basic_primitive()         [F1]
```

These two instruction call for a primitive instruction on the robot itself [F0] or on an other specific robot [F1].

```
   | if (test) { local_instruction }
     else { local_instruction }          [F2]
   | while (test) { local_instruction }  [F3]
   | loop local_instruction end loop     [F4]
   | exit                                [F5]
   | variable_assignement                [F6]
```

These instructions are the classical constructions for condition, assignement or loops inside a agent.

```
| emit event                            [F7]
```

Is used to emit an event. This event can be a social event if declared at the social level (it is then broadcast to all agents of the application) or a protected event (it is then shared by a group a agent in a **parofseq**). An event can also be used locally inside an agent behaviour to program some reactive architecture. The react part [D6] are local response to events.

```
| resume                                [F8]
| restart                               [F9]
| reevaluate                            [F10]
```

[F8] is used inside a react part and expresses that the next instruction to be executed it the one that was to be executed when an event was emitted.

[F9] restarts the *local_instruction* of the *where_with_event* [D3].

[F10] is an other powerful construction related to the **where** instruction. When reevaluate if executed then the control

moves back in the code to the first **where** of this block of program and starts the checking of the **where**(test) set of instruction. This is used to change the behaviour of a robot. If after an event, the robot modifies some internal state then it can now be member

IV. Example

In this section we show how Horocol can be used to express the behavior a real multi-agent system. The basic idea of this example is to show all the important features of the language. So we take a general example and will discuss in the next section the dependency on real robots.

Let's consider a school with 3 groups of students : A,B,C. They follow teaching courses in : Math, Sport, English and Puzzle. The rooms for teaching are numbered 101, 102 , 103 and stadium. The list of teachers is Marie (Puzzle), Betty (English), Georges (Sport), Albert (Math).

The schedule of the day is the following :

1	Group A & B Room 101 Math with Albert	Group C Stadium Sport with Georges	
2	Group A Room 101 English with Betty	Group B *PROG22* Room 102 Puzzle with Marie	GroupC Room 103 Math with Albert
3	Group A & B Stadium Sport with Georges*PROG31*	Group C Room 101 English whit Betty	

Table 1 : schedule of the day

In this example we will suppose that it exist two kinds of agents : Student and Teacher.

Social program

```
import Srobots.xml;
import Trobots.xml;
```

the real robots are define in xml files (see section IV)

```
programHorocol School
type student use  Srobots.xml ;
type teacher use  Trobots.xml ;
student a1, .... , a150  = newAgent(student);
teacher t1, t2, t3, t4 : = newAgent(teacher);
```

this is the declaration of 150 agents having type *student* and 4 agents having type *teacher*

```
{
a1.setIdentification ( "Bob", "A");
....
t1. setIdentification ( "Marie", "Puzzle"); ....
```

This is the initialisation part where we define the name of the student and his group and where the teacher are define with their name and the subject teach. We suppose that *setIdentification()* is a primitive define in both types *teacher* and *student* and implemented in the xml file.

```
|| ( [Prog11], [Prog12]);
|| ( [Prog21], [Prog22], [Prog23]);
|| ( [Prog31], [Prog32]);
}
```

This defines the three part of the day described in table 1.It is a sequence of team activities running in parallel. Here Prog11, .. Prog32 are team level programs in Horocol. We will now describe two of them : Prog22 (corresponding to the middle of the table 1 : Group B, room 102, Puzzle with Marie) and Prog31 (last line first row in table 1 Group A & B Stadium Sport with Georges).

Coordination programming

- **Prog22** : is an example to explain how we could implement a blackboard architecture [24]. We assume here that for the type student some primitives are defined : groupIs(), moveTo(), searchPiece(), myIdentification(), tryPieceOnPuzzle(). Same for the type teacher : moveTo (), subjectIs(), search(), addList().

```
parofseq (student, teacher){                    // Prog 22
 event handRaised( string x, int y) ;
 string studentAccessingPuzzle ;
```

This concern the two types : student and teacher. The execution will be parallel without synchronisation. One event is protected (shared by all agent running this part of code) is used by student to inform when they raise their hand. One variable is protected and is used to know the name of the student authorised to access to the puzzle (i.e. the blackboard)

```
 where (student.groupIs() = "B") {        // student code
   int pieceToTest ;
   moveTo(102);
   loop
    pieceToTest = searchPiece();
    emit handRaised (myIdentification(), pieceToTest) ;
    while(not(studentAccessingPuzzle = myIdentification())
{           wait() };
    tryPieceOnPuzzle (pieceToTest);
   end loop
 };
```

All students in group B will execute this code and will define locally a variable *pieceToTest*, this variable is duplicated in each agent. Each agent moves in the room 102 and begins the loop. First searches for a piece to try on the puzzle (here each agent is running its own code *searchPiece()* so the execution time can be very different from one to an other). When a piece is found the agent emit the protected event to inform the

teacher and wait for his turn, then tries his piece on the puzzle and search for a new one with the loop.

```
 where (teacher.subjectIs()="Puzzle" ) {      // teacher
code
    list waitingList ;
    moveTo(102);
    loop
     while ( not empty(waitingList)) {
       studentAccessingPuzzle = search(waitingList);
       wait();
     };
    end loop
  react
    when handRaising (X,Y) =>  addList(X,Y, waitingList);
                               resume;
  };
```

If the teacher speciality is "puzzle" then he executes this code (here we assume that only one teacher is having this speciality instead of what they would be several executions of this part of code). The teacher agent implements a local list to remember the list of students waiting to access to the puzzle. The agent moves in the room 102 and loops. He searches in the waiting list for a student to go at the puzzle and assign his name. Each time that an event *handRaising* is emitted then he memorise in the *waitingList* the name *X* of the student on the number *Y* of the piece, and continues its execution where it was interrupted. This construction can manage priority. The *search()* program can give a priority to someone in the *waitingList.*

In this example, the teacher and the students are making an active waiting (while loops). In Horocol, we could use **event** to awake some agent sleeping by the semantic of the **react** part of a **where** instruction.

```
}
```

end of the parofseq. Here going in sequence suppose that all the agents involved in the parofseq have terminate their code. If one of them is missing the program is blocked. We can by pass this problem by using specific **event** to kill process.

- **Prog31** : this part of the example shows how is used synchronous programming in Horocol. The idea here is to simulate the behavior of a group of people in a rowing boat. One (the teacher) is the leader giving the tempo and the students are divided in two groups : one on the left of the boat the other one on the rigth side. They must be a coordination of each movement of everybody.

To simplify we assume that they all are in "stadium" (part 1).

```
seqofpar(student, teacher){                    // Prog 31
```

Synchronous execution for all agents executing this part of code this means that each agent executes one instruction at the same time that all the others.

```
where ((student.groupIs()="A" | student.groupIs()="B") &
student.isOdd()) {
     loop   upLeft();          // 1
            moveLeftFront();   // 2
            downLeft();        // 3
            moveLeftBack();    // 4
     end loop
};
```

Agents concerned by this code must be student in group A or B and on the left of the boat.

```
where ((student.groupIs()="A" | student.groupIs()="B") &
not student.isOdd()) {
     loop   upRight();         // 5
            moveRightFront();  // 6
            downRight ();      // 7
            moveRightBack();   // 8
     end loop
};
```

Agents concerned by this code must be student in group A or B and not on the left of the boat.

```
where (teacher.subjectIs()="sport") {
     loop
            say("Hi");         // 9
            say("Ho");         // 10
     end loop
 };
}
```

In this last part we select the teacher (again we assume that there is only one in the boat).
The execution will be decomposed has follow:

- step 1 instrucions 1 and 5 and 9
- step 2 instructions 2 and 6 and 10
- step 3 instrucions 3 and 7 and 9
- step 4 instruciotns 4 and 5 and 10

and start again.

This kind of construction is very important in the field of reconfigurable robots where we need to express that all modules of the general structure are performing their actions simultaneously.
This example could be programmed by other ways but we retrain this presentation to show how the communication between agents can be treated in Horocol. This can implement constructions like in [14].

V. 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 or variables 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 or variable assignment.

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, light sensor no communication (think of a Lego Mindstorm robot) then constructions like : seqofpar or 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 uses the information included in the XML file. This file is including three levels of information :

- the *robot primitive* or local public variables specification,
- the *syntax of the language* used to program this robot,
- the *horocol system* primitives available 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, variable assignment.
The second phase is to check if all the primitives used in the Horocol program for the associated type are present in the *robot primitive*.
Finally a purely syntactic rewriting transformation is performed from the Horocol source code to the specific robot language. Of course this last pass is specific to each robot language so it needs to be rewrite for each kind of target. In our case we tested this transformation on our specific language developed in the Maam project [22].

VI. Conclusion

The Horocol language proposed here allows the description of multi-agents, multi robots behaviour at three different levels : social, team and local. We define two levels of parallelism: one, for the social organisation, expressing how teams are distributed in an application and the second one inside a team. The originality of Horocol is at this team level in the instructions : **parofseq/seqofpar,** for synchronous or asynchronous programming, coupled to the **where** instruction for precondition evaluation coming with the **reevalute** to check for dynamical change of behaviour. From an Horocol specification program it is possible to rewrite the program in the effective language for a specific robot.

Acknowledgment

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

References

[1] 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.
[2] R. Alur & all, *"Hierarchical Hybrid Modeling of Embedded Systems."* Proceedings of EMSOFT'01: First Workshop on Embedded Software, October 8-10, 2001
[3] J. Armstrong " The development in Erlang", ACM sighpla international conference on functional programming p 196-203. 1997
[4] M.S. Atkin & all "HAC : a unified view of reactive and deliberative activity. Notes of the European conf on artificial intelligence 1999
[5] M. Dastani & L. van der Torre " Programming Boid-Plan agents deliberating about conflicts along defeasible mental attitudes and plans" AAMAS 2003
[6] C. Gueganno and D. Duhaut "A hardware/software architecture for the control of self reconfigurable robots" DARS 04 7th symposium on distributed autonomous robotics systems, june 23-25, Toulouse France.
[7] M. Jorgensen & all "Modular ATRON : modules for a self-reconfigurable robot" IEEE/RSJ int conf on intelligent robots and systems IROS 2004 Sendai Japan
[8] P. Hudak & all " Arrows, robots, and functional reactive programming" lecture note in computer scinece 159-187 Spinger Verlag 2002
[9] 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
[10] E. Klavins " A formal model of a multi-robot control and communication task" IEEE conf on decision and control, 2003
[11] E. Klavins "A language for modeling and programming cooperative control systems" Int conf on robotics and automation ICRA 2004
[12] G. King "Tapir : the evolution of an agent control language" American association of artificial intelligence 2002.
[13] T. Lozano-Perez & R. Brooks "An approach toautomatic robot programming" Proceedings of the 1986 ACM fourteeth annual conf on computer scinec 1986, ACM Press
[14] D.C. Mackenzie & R. Arkin "Multiagent mission specification and execution" Autonomous robot vol 1 num 25 1997
[15] F. Mondada & all "Swarm-bot : for concept to implementation", IEEE/RSJ int conf on intelligent robots and systems IROS 2003
[16] 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
[17] I. Pembeci & G. Hager " A comparative review of robot programming languages" report CIRL – Johns Hopkins University august 14, 2001
[18] J. Peterson & all " A language for declarative robotic programming" Int conf on robotics and automation ICRA 1999
[19] T. Vu & all "Monad: a flexible architecture for multi-agent control" AAMAS'03
[20] E.Yoshida & all "Planning behaviors of modular robots with coherent structure issing randomized method" DARS 04 7th symposium on distributed autonomous robotics systems, June 23-25, Toulouse France.
[21] C. Zielinski "Programming and control of multi-robot systems" Conf. On control and automation robotics and vision ICRARCV'2000 dec 5-8 200à, Singapore
[22] http://www.univ-ubs.fr/valoria/duhaut/maam.
[23] http ://www.robocup.org
[24] Barbara Hayes-Roth home page: http://ksl-web.stanford.edu/people/bhr/