Learning Concurrent Programming: A Constructionist Approach

Giuseppina Capretti, Maria Rita Laganà, and Laura Ricci

Università degli Studi di Pisa, Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italia
{capretti,lagana,ricci}@di.unipi.it

Abstract. We present a software environment in which students learn concurrency by programming the behaviour of a set of interacting agents. The language defined puts together the turtle primitives of the Logo language, the classic sequential imperative language constructs and the concurrent ones. It is possible to program a dynamic world in which independent agents interact with one another through the exchange of messages.

1 Introduction

Today computer science is very important in society. The new working environments require an interaction with computers that, today, are the new medium through which ideas are expressed and programming is the means by which this can be accomplished. For this reason, learning to program has become more and more important for young people.

The concurrent programming paradigm is particularly interesting because of the diffusion of intrinsic parallel systems such as the massively parallel ones and networks. Our teaching experience shows that the learning of concurrent programming is often hard for students. We believe that this is due to the fact that the concurrency paradigm is often considered only as an evolution of the sequential one and, furthermore, students learn it after having acquired the latter.

This situation can be overcome if the first experience in programming is acquired through the concurrent paradigm at an early age. The concept of concurrency is intrinsic in the real world because it is *naturally* concurrent. If we give a student a concurrent programming environment, she/he may create and simulate all the fantastic and real situations s/he wants.

The theory underlying this approach is the constructionism [7]. Papert, the father of this theory, believes that students will be more deeply involved in their learning if they are constructing something that others will see, judge, and perhaps use. Through that construction, students will face complex issues, and they will make the effort to solve the problem and learn because they are motivated by the construction. A student should not be considered like a bottle to fill with notions but as a self-constructor of his/her knowledge: students need instruments to make the assimilation of new schemes and personal re-elaboration easier.

Our idea is to create a new system allowing the definition of the behaviour of automata able to interact with one another and react to the outside stimuli. In this case the term "animate system" is perfect to describe this sort of dynamic world that involves active and inter-active objects.

V. Malyshkin (Ed.): PaCT-99, LNCS 1662, pp. 200-206, 1999.

Animate systems are simulated dynamic worlds that contain multiple independent but interacting graphic actors [10]

The environment we are describing in this paper allows the definition of animated concurrent worlds programmable by an agent-based programming language.

An agent is any component of a program or system that is designed to be animated. The term agent - asserts Travers in [10] - suggests a variety of attributes that have not generally been built into the underlying metaphors of existing programming languages, attributes such as autonomy, purposefulness and the ability to react to the surrounding environment. In our case the creatures are autonomous and have the ability to react to the outside stimuli.

We have analysed several educational systems: KidSim [9], Starlogo [8] and ToonTalk [6]. We think that even if these systems are concurrent, they are not suitable for teaching concurrency.

The **Orespics** system, developed at the Computer Science Department of the University of Pisa [1, 2, 3] is programmable with Logo-PL language.

Logo-PL has control flow, movement and communication commands and expressions. The Logo-PL language defines a set of communication primitives. In particular, the prototype version implements basic primitives to send and receive messages. The receive primitive is synchronous and asymmetric while the send primitive is synchronous and symmetric [5]. As you can see, in this prototype, the set of the communication primitives is extremely poor.

In this paper, we introduce our new system, Advanced Orespics, its Orespics-PL language that includes a richer set of communication primitives and we propose an example of didactic training to learn the semantics of different types of communications.

2 The Advanced Orespics System

The Advanced Orespics system is born from the Orespics system described above. Its programming language is called Orespics-PL and is based on the local environment model and on the explicit use of the communication primitives. The Advanced Orespics system has substituted the previous one because a richer set of communication primitives is defined, activation and termination constructs are introduced and no limit to the number of interacting actors in the world is imposed. Each actor is an agent of an animate system and has the attributes of autonomy, purposefulness and the ability to react to the surrounding environment by the exchange messages paradigm. An agent is characterised by a set of properties: the initial position on the screen, its appearance and the code of its program

The system gives the users an interface to define all these properties. The system has a set of pre-defined fantastic and real characters like aliens and animals. The students may choose the most suitable character according to the situation to solve.

The sequential part of Orespics-PL includes traditional imperative sequential constructs (*repeat*, *while*, *if* ...) and all turtle primitives of the Logo language [4]. Orespics-PL language offers all the elementary data types (integer, boolean..): the only data structure is the list. Some of the operations defined on list type are *getFirst(list), first(list)* and *second(list)*: *getFirst* returns the first item of *list* and pops it up; *first* and *second* return respectively the first and the second ones and do not pop them up.

The set of primitives, functions and procedures used in the following examples are:

- *versus(x, y)*, which returns the direction to assume to reach the point of co-ordinates *x* and *y*,
- *distance(x, y)*, which returns the distance between the position of agent and the point *(x, y)*,
- *set_heading(angle)*, which turns the agent in the direction given by the *angle*,
- *set_color(color)*, which sets the colour of the agent trace to that of *colour*. A set of pre-defined colour is available,
- *jump(x, y)*, the agent jumps to the point of co-ordinates *(x, y)*,
- *random(val)*, which returns a random value included in +/- *val*. If parameter is zero, it returns a random value according to the common definition.

As regards the concurrent part, the new language defines the following types of primitives:

- synchronous send and receive,
- asynchronous send and receive
- termination and activation commands,
- broadcast send primitive,
- asymmetric receive primitive.

The didactic training we present in this paper uses only synchronous and asynchronous communication primitives: hence, we only show the syntax and the semantics of these primitives. The syntax of the synchronous primitives is changed in:

send&wait *msg* **to** *agent*
wait&receive *var* **from** *agent*

The semantics of synchronous and asynchronous primitives is well known in literature [5]. With regard to the synchronous primitives, when an agent sends a message to another one and its partner is not ready for communication, it waits until the message has been received. The semantics of synchronous receive primitive is analogous.

The syntax of the asynchronous primitives is:

receive&no_wait *var* **from** *agent*
send&no_wait *msg* **to** *agent*

As for the asynchronous primitives, an agent sends/receives a message to/from another one but it does not wait for the successful issue of the communication. When an agent executes a *send&no_wait*, it does not wait for the receiver to get the message and it goes on with its execution. Hence a queue of messages is created, where messages are inserted and taken according to the order of arrival. When an agent executes a *receive&no_wait* it checks the existence of some incoming messages and goes on. If the queue is empty the message has no meaning and no value is assigned to the *var*. The meaning of *var* may be checked through the function *in_message()* which returns the **true** value if the last executed *receive&no_wait* has picked up a valid message, and the **false** value otherwise.

A process executing the *receive&no_wait* performs a non-deterministic choice: we suppose that a suitable introduction of non-determinism in concurrent programs has been given when this primitive is introduced to the students.

3 Teaching Synchronous and Asynchronous Communication

We now wish to introduce an example of didactic training to learn the semantics of the communication paradigm we described above. In particular the goal of the training is to allow the students to learn the difference among different kinds of communication.
According to the constructionist approach, this is obtained by proposing a set of proper problems to the students and letting them solve them in the way they prefer. The whole process is obviously guided by the teacher; nevertheless, the student may try different solutions and verify the effects of his program directly on the screen, eventually changing the program. We present only the code of the agents used to solve the problems proposed: we suppose they are just created and that every one of them has properly been defined.

The task of the first problem is the comprehension of the concept of synchronisation among agents. Let us consider the following problem:

"*The Quasar 40 and the Star 23 space shuttles move about the solar system. They have to meet to transfer the crews of Star-23 to Quasar-40 for the party for the captain of the latter. They use the radio communication to exchange the position. Try to describe their behaviour.*"

To define a correct synchronisation between these two agents, it is necessary that each of them knows the behaviour of the other one; in particular, if and when the other one is disposed to accept synchronisation. In the following solution, the two agents[1] employ synchronous send and receive to exchange their positions: this is probably the first solution given by the students, but it is not correct.

```
Agent Quasar_40
   repeat
      x ← random(25);
      y ← random(15);
      right x;
      forward y;
      send&wait [myX, myY] to
      Star_23;
      wait&receive [x, y] from
      Star_23;
      d ← distance(x, y);
   until d < 10;
   show "It's time to stop";
end
```

```
Agent Star_23
   repeat
      x ← random(25);
      y ← random(15);
      right x;
      forward y;
      send&wait [myX, myY] to
      Quasar_40;
      wait&receive [x,y] from
      Quasar_40;
      d ← distance(x, y);
   until d < 10;
   show "It's time to stop";
end
```

The variables *myX* and *myY* are predefined variables that represent the position of the agent on the screen.

This kind of solution determines a deadlock situation between agents. In fact, the agents move and execute a *send&wait*, waiting for the reception of the corresponding message: the agents are now blocked. It is possible to use this example to deal with the deadlock subject and discuss the semantics of prevention, avoidance, recognition or elimination of the deadlock with students. Our system may help teachers to create situations in which the deadlock is not a difficult concept, but only one of the innate characteristics of a concurrent environment. To solve the problem of the deadlock it is sufficient to invert the order of send and receive in the code of one of the agent

[1] Each agent represents a space shuttle

```
Agent Quasar_40
……
      forward y;
      send&wait [myX, myY] to Star_23;
      wait&receive [x, y] from Star_23;
      d ← distance(x, y);
……
end
```

```
Agent Star_23
……
      forward y;
      wait&receive [x, y] from
      Quasar_40;
      send&wait [myX, myY] to Quasar_40;
      d ← distance(x, y); ……
end
```

In the above examples we use synchronous primitives. We could also propose to solve the deadlock problem through asynchronous primitives: the following is a possible solution.

```
Agent Quasar_40
……
      forward y;
      send&no_wait [myX, myY] to
      Star_23;
      wait&receive [x, y] from Star_23;
      d ← distance(x, y);
……
end
```

```
Agent Star_23
……
      forward y;
      send&no_wait [myX, myY] to
      Quasar_40;
      wait&receive [x, y] from
      Quasar_40;
      d ← distance(x, y); ……
end
```

It is important to notice that the *receive* command must be synchronous in this case because each shuttle needs the message from the other one before computing the distance. This version is similar to the one described previously, but it presents no deadlock. The comparison between the two versions can be used to study and probe the semantics of the two kinds of primitives.

The concept underlying the following problem is the understanding of the asynchronous communication. We propose a problem requiring only that the messages exchanged be picked up from the receiver according to the arrival order.

The problem proposed is:

"*A leader pen orders the red pen to draw the roof of a house and the black pen to draw the walls. It gives them the co-ordinates of the points. The coloured pens draw the segments creating the roof and the walls of the house. How do they do it?*".

```
Agent leader_pen
    walls ←[[5,9], [5,4], [10,2],
    [10,7], [14,7], [14,2], [10,2]];
    roof ← [[14,7], [12,10], [10,7],
    [5,9], [8,11], [12,10]];
    repeat
        send&no_wait getFist(walls) to
        black_pen;
    until (walls = nil);
    send&no_wait nil to black_pen;
    repeat
        send&no_wait getFirst(roof) to
        red_pen;
    until (roof = nil);
    send&no_wait nil to red_pen;
end
```

```
Agent black_pen
    wait&receive [x, y] from leader_pen;
    jump(x, y); set_color(black);
    pen_down:
    repeat
        wait&receive msg from
        leader_pen;
        if msg <> nil
        then
            x ← first(msg);
            y ← second(msg);
            set_heading (versus(x, y));
            forward (distance (x, y));
        endif
    until (msg = nil)
end
```

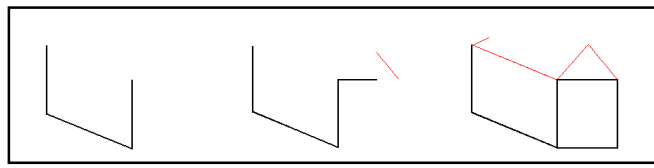

Fig. 1 Evolution of drawing: first case

The code of *red_pen* is identical to that of the *black_pen*. In this version the *leader_pen* agent first sends all the walls' points to *black_pen* and then the list of the roof points to *red_pen*. In fig. 1, we show a possible evolution of the drawing: the black line is thicker than the red one to grant the visual identification of the lines.

In the following version, *leader_pen* alternatively sends a point to *red_pen* and one to *black_pen*. The code of coloured pens is identical to the precedent while the one of *leader_pen* is modified as follow:

```
Agent leader_pen
    walls ←[[5,9], [5,4], [10,2],
    [10,7], [14,7], [14,2], [10,2]];
    roof ← [[14,7], [12,10], [10,7],
    [5,9], [8,11], [12,10]];
    repeat
        if (walls <> nil)
          send&no_wait getFirst(walls)
          to pen_black;
        endif
        if (roof <> nil)
          send&no_wait getFirst(roof)
          to pen_red;
        endif
    until ((roof=nil) AND
    (walls=nil))
    send&no_wait nil to pen_red;
    send&no_wait nil to pen_black;
end
```

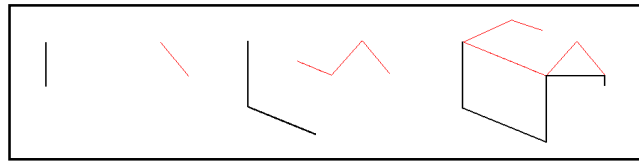

Fig. 2 Evolution of drawing: second case

The purpose of the last example is to show that a realistic situation, like the modelling of animal behaviour, may be programmed in our system. In this example both *send&no_wait* and *receive&no_wait* primitives are used. Since the latter primitive requires a clear knowledge of the concept of non-deterministic behaviour of an agent, we suppose it has been given in another didactic training.

"In a field there were two ants called Z ant and T ant. They are searching for food and have a term: the first who finds it tells the other the position of food. "Good luck T". "Good luck Z. Let us begin"

We present one of the possible implementation of the Z ant and T ant agents.

```
Agent Z_ant
    I_found ← false; You_found ← false;
    repeat
        x ← random(25);
        y ← random(15);
        right x; forward y;
        if here_food(myX, myY)
        then
            send&no_wait [myX, myY] to
            T_ant;
            I_found ← true;
        else
            receive&no_wait [x, y] from
            T_ant;
            if in_message()
            then You_found ← true; endif
        endif
    until (I_found OR You_found);
    if You_found
    then
        set_heading (versus(x, y));
        forward distance(x, y);
    endif
end
```

```
Agent T_ant
    I_found ← false; You_found ← false;
    repeat
        x ← random(25);
        y ← random(15);
        right x; forward y;
        if here_food(myX, myY)
        then
            send&no_wait [myX, myY] to
            Z_ant;
            I_found ← true;
        else
            receive&no_wait [x, y] from
            Z_ant;
            if in_message()
            then You_found ← true; endif
        endif
    until (I_found OR You_found);
    if You_found
    then
        set_heading (versus(x, y));
        forward (distance(x, y));
    endif;
end
```

Each ant moves randomly checking for presence of food. If an ant finds it, it sends the food co-ordinates to the other one and stops moving. If it does not find it, it checks the presence of an incoming message from the other ant. If no message is present it goes on searching. If a message is present it gets the co-ordinates to reach the food.

It is important to stress that in this case the agents are completely autonomous. Each one may be programmed without knowing the behaviour of the other agent: this is the main difference between this example and the previous ones. The autonomy increase is obtained through the use of the *receive&no_wait* primitive, which allows each agent to perform a non-deterministic choice whose result depends on its interaction with the surrounding world.

4 Conclusions

We may create lots of new examples in which the agents co-ordinate and synchronise themselves. Classic problems like the game of life, the simulation of a biological system, and so on may be naturally realised in our system.

We have studied the possibility of creating several typologies of agents characterised by a richer set of personal properties, for example, in the case of the ants we give them the ability to move the antennae or in the case of the dog the ability to bark. We are implementing a version in which the character may be created or imported by the student.

We are just planning experimentation of the Advanced Orespics system with teenagers to test the suitability of the Orespics-PL language.

References

1. Capretti G.: *Strumenti per l'apprendimento della concorrenza nella didattica dell'informatica*, Master of Computer Science Thesis, Computer Science Department, University of Pisa, December 1997.
2. Capretti G., Cisternino A., Laganà M. R. and Ricci L.: A concurrent microworld, *ED-MEDIA 99 World Conference on Educational Multimedia, Hypermedia & Telecommunications*, Seattle, Washington, June 19-24th.
3. Capretti G., Laganà M. R. and Ricci L.: Micro-world to learn concurrency, *SSCC'98*, 22-24 September 1998, Durban, South Africa, 255-259.
4. Harvey B.: *Computer Science Logo style*, The Mit Press, Cambridge, 1997
5. Hoare C. A. R.: Communicating Sequential Process, *Comm. of the ACM*, Vol. 21, No. 8, Aug. 1978, pp. 666-677.
6. Kahn K.: ToonTalk - An Animated Programming Environment for Children, *Journal of Visual Languages and Computing*, (7), 197-217, 1996.
7. Papert S.: *Mindstorm: children, computer and powerful ideas*, Basic Books, New York, 1980.
8. Resnick M.: *Turtles, termites and traffic jam: exploration in massively parallel micro-world*, The MIT Press, Cambridge, 1990.
9. Smith D. C. and Cypher A.: KidSim: end users programming of simulation, Apple Computer Inc., 1997. **http://www.acm.org/sigchi/chi95/Electronic/documnts/papers/ac1bdy.htm**.
10. Travers M. D.: *Programming with agents: new metaphors for thinking about computation*, Bachelor of Science Thesis at Massachusetts Institute of Technology, 1996.