

Experiences using Ada in a Real-Time and Distributed laboratory

P. Balbastre

patricia@disca.upv.es

S. Terrasa

sterrasa@disca.upv.es

J. Vila

jvila@disca.upv.es

A. Crespo

alfons@disca.upv.es

Departamento de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia

Spain

1. ABSTRACT

This paper reviews our experience in the laboratory of two courses in the Control Engineering Curriculum at the Politechnical University of Valencia: Real-time Systems and Distributed Control Systems. These courses, in a Control Engineering curriculum, require the development of industrial prototypes to validate different aspects of the system development: control algorithms, real-time software design, external devices integration, distributed control, etc. To meet all these goals a robot guidance experience has been selected. In this paper, we describe the experience in the design and implementation of a virtual and real environment developed in Ada to complete the student laboratory activities. The system is complex enough to allow the students to analyze, design and implement a complete prototype

1.1 Keywords

Real-Time systems, distributed systems, robot application.

2. INTRODUCTION

The importance of real-time systems is rapidly growing and so, its role has to be consolidated in the control engineer curriculum where it is key for many subjects. The objective of real-time courses is to discuss topics and problems encountered when developing embedded and distributed systems for hard and soft real-time environments. Industrial application have to used as basis for the experimentation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGAda'99 10/99 Redondo Beach, CA, USA

© 1999 ACM 1-58113-127-5/99/0010...\$5.00

From the educational point of view, the Ada language provides excellent capabilities to be used as language to develop concurrent and distributed real-time applications. This way, the language provides all the normal constructs of high-level languages as well as low-level features. In addition, Ada also offers many features over other standard languages, such as packaging in modules, data abstraction, encapsulation, object-oriented features, and well-defined real-time multitasking can be exploited.

The use of complex industrial applications, like robotic systems provides an excellent example to cover the main objectives of the curriculum. In this sense, a guided vehicle has relevant control aspects, real-time constraints, external access to devices, and distribution possibilities.

This paper reviews our experiences in the laboratory experimentation of two courses in the Control Engineering Curriculum at the Politechnical University of Valencia. The first is the Real-time systems. In this course, we are using Ada's ability to design and implement real-time applications. The second course is the Distributed Control Systems. In this course we extend Ada applications to be executed in a distributed environment.

These courses, in a Control Engineering curriculum, require the development of industrial prototypes to validate different aspects of the system development: control algorithms, real-time software design, external devices integration, distributed control, etc. To accomplish these objectives a common set of processes have been selected to be used by the real time courses: a pilot plant of residual water and a guided vehicle. The students have to design the software and hardware required to implement an embedded and a distributed control system of both processes. In this paper we describe the design, implementation and student activities of a guided vehicle. The system is complex enough to allow the students to analyze, design and implement a complete prototype.

This paper is organized as follows, in section 3 we describe the objectives of both courses. Section 4 details the main robot features and functionality. Section 5 shows the complete design and the Ada interfaces. Section 6 details

the proposed implementation. Section 7 reviews the criteria for distribution. Finally, some conclusions are stated.

3. REAL-TIME SYSTEMS AND DISTRIBUTED CONTROL SYSTEMS.

Then Real-Time Systems and Distributed Control System courses are included in the Control Engineering Curriculum of the Politechnical University of Valencia [1]. The control engineer has to be able to design and implement control systems. The control courses provide the knowledge and abilities to understand and analyze industrial processes and the methods and techniques to design and implement the control system. The control system engineer should be specialized in many aspects, such as:

- Control theory
- Modeling
- Simulation
- Advanced control techniques
- Real-time system design
- Programming general purpose or specific devices
- Networking
- Planning (CIM applications)
-

From the real time point of view, the main objectives are based in some recommendations [2]:

- To understand the problems related to the design and analysis computer control system and the timing constraints.
- To design and implement multi-tasking applications using high level languages involving several control loops and data storage and representation.
- To acquire the ability to design and implement distributed applications
- To produce specific embedded systems

The course Real-Time systems is the basic course [3] and has as concrete objectives:

- To understand the timing constraints and the elements that have influence in the development of computer control applications.
- To specify the system behavior by means of Petri nets
- To design and implement computer control applications
- To know the high level languages and their abstractions to implement concurrency, communication and time control.
- To know the influence of the operating system and the scheduling algorithms
- To analyze the techniques for schedulability analysis.

The Distributed control systems has the following objectives:

- To describe the basic network architectures and protocols for industrial control systems.
- To review the different network technologies and performance, and their scope of application
- To describe the operating system services and protocols programming interface to develop distributed applications.
- To introduce the architectural models and language support for designing distributed control applications.
- To provide the ability for designing distributed control applications.

4. PROCESS DESCRIPTION AND FUNCTIONALITIES

The process to be controlled in the laboratory is a small Khepera robot [4]. It measures 27mm of radio and 2cm of thickness (Figure 1). It has two wheels, which are controlled by two DC motors. Furthermore, these wheels have each one an incremental encoder supplying odometric information. This odometric information will be used to calculate the robot position.

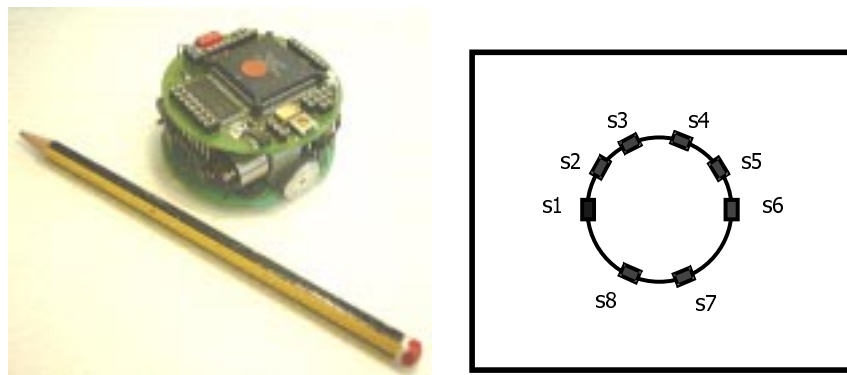


Figure 1. Robot picture and sensor layout

The sensor system is formed by eight infrared sensors, capable measuring distances (scope 60mm) and environmental light. Its location along the robot is three sensors in the front left side (10°, 45° and 85°); three in the front right side (-10°, -45° and -85°) and two sensors in the back side (35° and -35°). Right hand side of figure 1 shows the topology distribution of the sensor in the robot.

The control of the robot requires to determine its turn. This turn is achieved by controlling the speed of each wheel. If the speed of the two wheels is the same, then the robot goes ahead. If it is different, then the robot turns. The turn is determined by the application functionality. Robot operation can be classified into two modes (Figure 2): robot chase (Figure 2b) and obstacle avoidance (Figure 2a). In both cases the control system has to achieve the following requirements:

- The system permits two operational modes: real or simulated. In the simulated mode, the scenario and the robots are virtual. In the real mode, each robot is a real-world component (Khepera).
- In any robot operational mode, the controller has to avoid the obstacles (static or another robots) founded in the robot path. In order to avoid obstacles, the robot controller implements the Braitenberg algorithm [5].
- When the simulated mode is defined, the scenario (arena and static obstacles) has to be defined by the user.
- The operator can, at any moment, start or stop any of the robots.

Now, we are going to describe the two operations mode.

4.1 Obstacle avoidance

In this case, the robot has to sequentially follow several goals that are defined by the user. Goals can be either defined in advance (all in a row) or after reaching the current goal.

The turn calculation takes into account two behaviours or decisions: heading for the destination and avoiding

obstacles. Both behaviours contribute with a turn and the final turn is a function of these two turns. It responds to the next expression:

$$Turn = Destination_turn * Destination_decision + Avoiding_turn * Avoiding_decision$$

Destination_decision and Avoiding_decision are factors that represent the importance of these two behaviours. Obviously, when there are no obstacles close to the robot, the avoiding factor will be null, and it will increase when an obstacle becomes closer. The destination factor will also take its maximum value (of 1) when there are no obstacles. In an intermediate case, the factors would take values between 0 and 1. The sum of these two factors is the unity. The calculation of the turn regarding the destination decision matches the next expression:

$$Absolute_turn = arctan \frac{y_robot - y_destination}{x_destination - x_robot}$$

The calculation of the turn regarding the avoiding decision is based on the Braitenberg algorithm:

$$Avoiding_turn = k_1 s_1 + k_2 s_2 + k_3 s_3 + k_4 s_4 + k_5 s_5 + k_6 s_6$$

where $k_6 = -k_1$, $k_5 = -k_2$, $k_4 = -k_3$ and s_i are the sensors reading.

4.2 Robot chase

In this mode there are two robots: the leader robot and the follower robot.

The leader robot generates a quasi-random trajectory. The follower robot has to pursue the leader, so this robot represents a dynamic goal. The trajectory of the follower robot can be described by the straight line that links a robot with his leader every sample period.

One robot can be the follower of a leader. At the same time, it can also be the leader of another robot. In this way there can be many leaders and many follower robots.

In this operation mode, dynamic goals make the control system more complex. For the leader, a dynamic goal is the result of generating a quasi-random trajectory. This goal must change frequently in order to avoid the follower robot

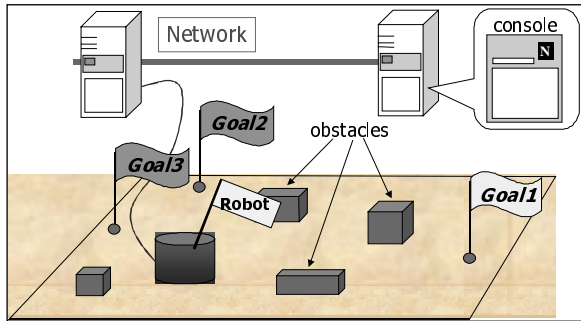


Figure 2 (a) Obstacle avoidance

to catch the leader robot. Nevertheless, the dynamic goal can not change too often because the control action could result in consecutive sudden turns. For the follower robot, the dynamic goal is the position of the leader, so both robots (leader and follower) will be treated as the same objects, following a dynamic goal.

The position of all robots is calculated with the odometric information provided by two incremental encoders (one for every wheel). Another possibility could be replacing the encoders information by putting a zenithal camera over the arena. In this way, we have an exact position of each robot and it is avoided the problem of cumulative errors with the previous solution.

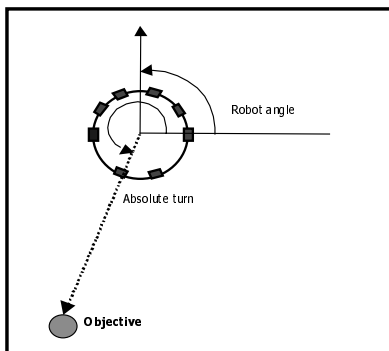


Figure 3. Experiment scheme

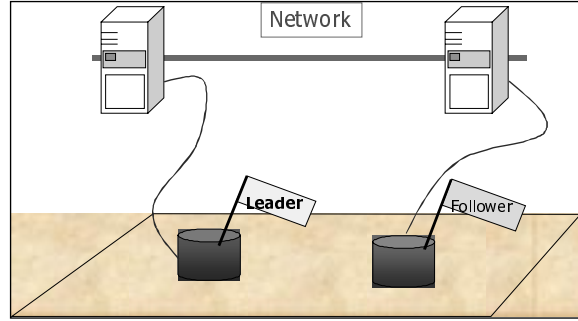


Figure 2 (b) Robot Chase

5. ARCHITECTURAL DESIGN

The architectural design of the global system is presented and discussed with the students. During the lectures, they have acquired the knowledge about the methodology and notation. Finally, the architectural design, showed in the Figure 4, is considered for implementation.

The design has to take into account the operation with several robots running in the same arena as well as the possibility of distribution.

Initially three main components are considered:

- **User_interface:** It is an interface to command the robots operation and displaying the situation of the robots in the arena. The Operator task gets the operator commands and sends the proper messages to the other components of the design. The Monitor is a periodic task that displays the arena with the obstacles and the robot situation through a graphical interface.
- **Robot_Controller:** This package encapsulates the control of one of the robots. The control of a robot is accomplished through the definition of three periodic tasks: Sensor, Control and Supervisor. All these tasks share a package that provides the global status of the robot. The Sensor tasks samples periodically a sensor data and stores it in the Robot_status object. The

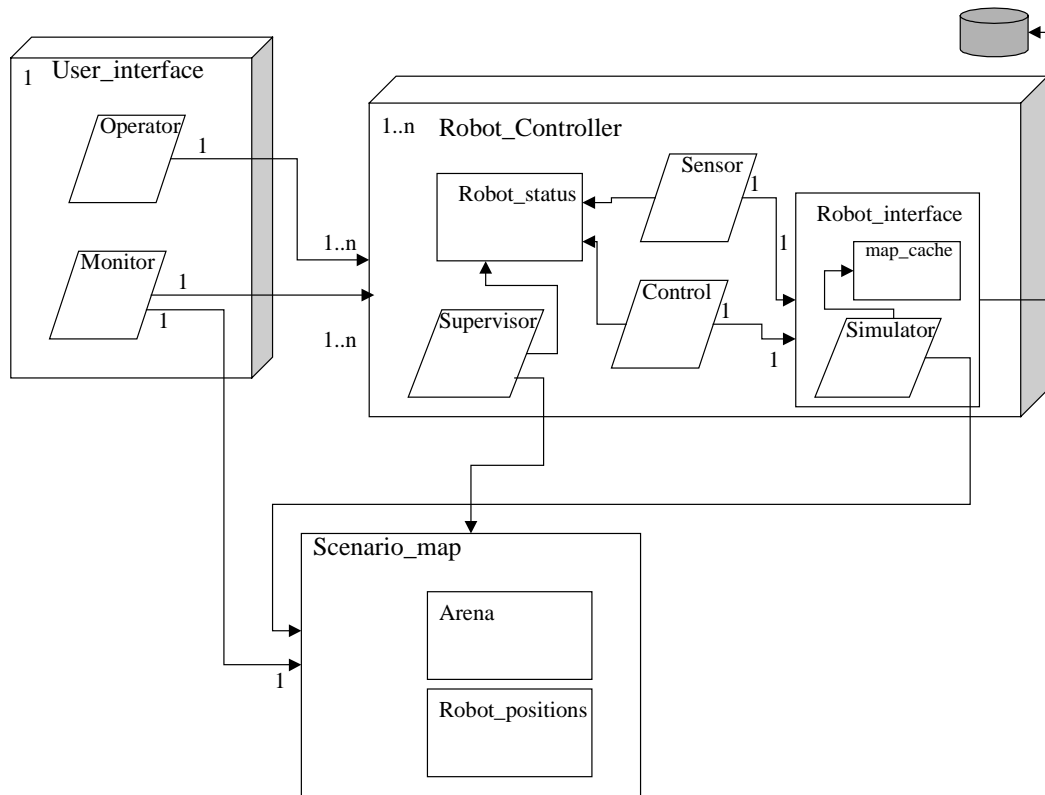


Figure 4. Architectural scheme

Control task reads the sensor information and determines the wheel speeds considering the obstacle avoidance algorithm and the goal to be reached. The Supervisor task supplies consecutive goals to be reached. In the goal oriented mode, these goals are queued by the operator and the supervisor detects when a goal has been reached and provides the next one. In the robot chase mode, the goal is dynamic being the result of the position of its robot leader. The Robot_interface package offers a unique robot abstraction to manage both the virtual and real environment. A internal task (Simulator) simulates the real robot when the virtual mode is selected. There will be as many instances of the Robot_Controller as robots are defined. Each instance can be executed in a different node when a distributed system is considered.

- Scenario_map: This package keeps the information about the real/simulated scenario (arena and obstacles) and the robots running in it. Periodically this information is displayed in the graphical interface.

6. IMPLEMENTATION

This section describes how the main components of the system are implemented. The students start from the architectural design and the package specification. The specification packages and the description of the tasks is summarized in the following. In the next points, Ada

implementation is described. Next, some notes of Java implementation are presented in section 6.4.

6.1 Robot Controller

The Robot Controller provides the abstraction to operate a robot. It uses a Global_Types package that defines several common types. The package specification has the following definition:

```
package Global_Types is
  Max_Robots: constant := 5;
  type Execution_Mode is (Virtual, Real);
  type Operation_Mode is (Goal_Oriented, Robot_Chase);
  type Position is
    record
      X_Pos,Y_Pos : Float;
      Angle: Float;
    end record;
  type All_Positions is array (1..Max_Robots) of Position;
  type Robot_Id is new Integer range 1..Max_Robots;
end Global_Types;
```

```
with Global_Types; use Global_Types;
with Robots_Status; use Robots_Status;
with Queues; use Queues;
package Robots_Controller is
  protected type Robot_Controller is
    procedure Define_Robot(Id: in Robot_Id);
    procedure Define_Execution_Mode(M:in Execution_Mode);
```

```

procedure Define_Operation_Mode(M:in Operation_Mode);
procedure Define_Goal(G: in Position);
procedure You_Are_Leader;
procedure Your_Leader_Is(Leader: in Robot_Id);
procedure Start_Robot;
procedure Stop_Robot;
procedure Initial_Position(Pos: in Position);
private
My_Identification : Robot_Id;
Execution : Execution_Mode;
Operation : Operation_Mode;
My_Leader : Robot_Id;
RStatus : Robot_Status;
Goals : Queue;
end Robot_Controller;
end Robots_Controller;

```

The Robot_Controller implementation defines three tasks: Sensor, Control and Supervisor. All these periodic tasks present the same interface in order to start, stop and finish, and share the state of the robot through the Robot_status package. As example, we detail the Control interface.

```

task Control is
  entry Start;
  entry Stop;
  entry Finish;
end Control;

```

6.1.1 Sensor task

The Sensor task obtains information of the sensors, provided by the Robot_interface, and stores it in the Robot_status. Sensors are accessed individually in each period. So to complete all the sensor information, the task has to read the eight sensors. As consequence, the period of this task is eight times faster than the Control task. On the other hand, this task also updates encoders values of the Robot_Status Package., values that can be read in the Control task.

6.1.2 Control task

The control task has to guide the robot to the goal. To determine the control action, the task consider the robot information (sensors, current_position, angle, motor speed) and the goal. Obstacles and other robots are seen through the sensor system. The control action is the wheel speeds. It determines the global robot speed and turn. To calculate the turn it has to considered two decisions: heading for the destination and avoiding obstacles. A basic algorithm to obstacle avoidance is provided to the students. The Control task body is :

```

task body Control is
  Period : Time_Span := Milliseconds(1200);
  Next : Time;
  Distance : Distance_Sensor;
  Pos : Position;
  Action : Decision;
  Turn : Float;

```

```

Go_Speed, Turn_Speed : Integer;
Aux_Sensors : Distance_Sensor;
P, G : Position;
I : Integer:=1;
begin
  Accept Start;
  Main_Loop : loop
    Next := Clock + Period;
    Control_Loop : loop
      -- First we get current abstract sensors values
      for J in 1..Max_Sensors loop
        Aux_Sensors(J):= Rstatus.Get_Sensor(J);
      end loop;
      -- Control action calculation
      Control_Algorithm(Aux_Sensors,
        Rstatus.Get_Position,
        Rstatus.Get_Goal, Turn, D);
      if (Turn>0.2) then Action:=Turn_Right;
      elsif (Turn<-0.2) then Action := Turn_Left;
      else Action := Go;
      end if;
      --Updates the decision vector
      D(I).D:=Action;
      D(I).P:=Rstatus.Get_Position;
      I:=(I mod 3)+1;
      Turn_Speed := Rstatus.Get_Turn_Speed;
      Go_Speed := Rstatus.Get_Go_Speed;
      case (Action) is
        when Turn_Right => Put_Right_Speed(Turn_Speed);
          Put_Left_Speed(-Turn_Speed);
        when Turn_Left =>
          Put_Right_Speed(-Turn_Speed);
          Put_Left_Speed(Turn_Speed);
        when Go => Put_Right_Speed(Go_Speed);
          Put_Left_Speed(Go_Speed);
        when Stop => Put_Right_Speed(0);
          Put_Left_Speed(0);
        when others => null;
      end case;
      select accept Stop; exit Control_Loop;
      or accept Finish; exit Main_Loop;
      or delay until Next; Next := Next + Period;
      end select;
    end loop Control_Loop;
    select accept Finish ; exit Main_Loop;
    or accept Start;
    end select;
  end loop Main_Loop;
end Control;

```

6.1.3 Supervisor task

The Supervisor task has a double goal: to prevent anomalous situations, and to establish the goals to be reached by the robot. To prevent abnormal functioning, it considers two different possibilities: i) the robot is too close to an obstacle and can crash, and ii) the robot is in a blocking situation. In both cases, the Supervisor will

modify (increase, reduce or stop) the robot speed depending on the situation. In block situations, when the robot is unable to reach an objective, a secondary destination will be set. Later, the control will attempt to reach the previous objective. In order to set a goal, the Supervisor will either consider the goals in the user defined order (goal oriented mode) or it will read, periodically, the leader position and fix its position as a new goal. Moreover, it is in charge of taking the decision of considering when a robot has reached a goal and then stop it. This task has a longer period than the other controller tasks, since it is supposed that the previous activities are less frequent than the control and data acquisition.

6.1.4 Robot_status package

The Robot_Status package maintains the robot state. The state is composed by several variables related to the robot and its operation.

```
with Global_types; use Global_Types;
with Robot_Components; use Robot_Components;
package Robots_Status is
protected type Robot_Status is
  procedure Put_Sensor(Sensor_Id:in Integer;
                      Sensor:in Abstract_sensor);
  function Get_Sensor(Sensor_Id:in Integer)
    return Abstract_sensor;
  procedure Put_Turn_Speed(turn_speed : in Integer);
  function Get_Turn_Speed return Integer;
  procedure Put_Go_Speed(go_speed : in Integer);
  function Get_Go_Speed return Integer;
  procedure Put_Position(Pos : in Position);
  function Get_Position return Position;
  procedure Put_Goal(Goal : in Position);
  function Get_Goal return Position;
  procedure Put_State (s : state);
  function Get_State return state;
  procedure Put_Mode (m: Execution_Mode);
  function Get_Mode return Execution_Mode;
end Robot_Status;
private
  Radius           : float := 26.0;
  Sensors          : Distance_Sensor;
  Mode             : Execution_Mode;
  Current_Turn_Speed : integer;
  Current_Go_Speed  : integer;
  Current_Position  : Position;
  Current_Goal      : Position;
  Current_State     : state;
end Robots_Status;
```

Robot_Components package contains all robot's components as it is shown in the next code:

```
with interfaces.c.extensions;
use interfaces.c.extensions;
with global_types; use global_types;
package Robot_Components is
  Max_Sensors : constant := 8;
```

```
Max_Speed : constant integer := 100;
Max_Encoder_Count : constant := 4294967295;
Reach : constant float :=60.0;
type decision is (turn_left, turn_right, go, stop);
type Simple_decision is
  record
    p : position;
    d : decision;
  end record;
type Decision_Vector is array (1..3)
  of Simple_Decision;
type Infrared_Sensor is array(1..Max_Sensors)
  of Integer;
type Abstract_Sensor is
  record
    distance : float;
    detector : boolean;
  end record;
type Distance_Sensor is array(1..Max_sensors)
  of abstract_sensor;
subtype Speed_Range IS integer
  range -Max_speed..Max_Speed;
subtype Encoder_Range IS long_long
  range 0 .. Max_Encoder_Count;
type Motor_Speed is
  record
    left, right : Speed_Range;
  end record;
type Encoders is
  record
    left, right : Encoder_Range;
  end record;
type state is (wait, stop, run);
end Robot_Components;
```

6.1.5 Robot_Interface package

This package provides the robot interface to a virtual or real robot. The operations allow to access to the robot sensors, encoders and wheel speed. As result of a control action, the Control task will fix the wheel speed.

The packages hides a Simulator task that permit to work with virtual robots. The Simulator is a periodic task implementing a physical model of the robot. This task generates the values of the encoders, wheel speed and sensors. In order to generate sensor information, the task uses a scenario_cache with the static information (arena and obstacles) and the other robot positions. Each period, the task reads the dynamic part of the scenario. The simulation could be very simple at the beginning and more complex when the students incorporate a model more detailed.

```
with Global_Types; use Global_Types;
with Robot_Components; use Robot_Components;
with Interfaces.c.extensions;
use Interfaces.c.extensions;
```

```

with Maps; use Maps;

package Robots_Interface is
  function Get_Next_Sensor(Id_Sens: in Integer)
    return integer;
  function Get_Left_Encoder  return long_long;
  function Get_Right_Encoder return long_long;
  function Get_Left_Speed  return Integer ;
  procedure Put_Left_Speed(S: in Integer);
  function Get_Right_Speed return Integer;
  procedure Put_Right_Speed(S: in Integer);
  function Get_Mode return Execution_Mode;
  procedure Put_Mode(M: in Execution_Mode);
  procedure Stop;
  procedure Start;
  procedure Finish;
  procedure Put_Map_Cache (B: Map; hr : in float;
                          wr :in float;
                          hs :in integer;
                          ws :in integer);
end Robots_Interface;

```

Although this package is shared by two tasks (Sensor and Control), it is not defined as protected. The internal state can be considered as the aggregation of two states that are accessed independently. While the Sensor task reads the sensors, the Control task reads and updates the rest of the variables. However, the robot access is made via serial channel, and it is necessary to serialize all messages that tasks will send to the robot. To perform this operation a serialized task is implemented .

```

-- Specification
task Serialized is
  pragma priority (1);
  entry Start;
  entry Read_Sensors(S : out Infrared_Sensor);
  entry Read_Encoder(E : out Encoders);
  entry Write_Speed(W: in Motor_Speed);
  entry Finish;
end Serialized;
--Implementation
task body Serialized is
  begin
    accept Start;
    Interface.Connection;
    Main_Loop: loop
      select
        accept Read_Sensors(S : out Infrared_Sensor)
          do Interface.Read_Sensors (S);
        end Read_Sensors;
      or
        accept Read_Encoder(E : out Encoders)
          do Interface.Read_Encoders(E);
        end Read_Encoder;
      or
        accept Write_Speed(W: in Motor_Speed)
          do Interface.Write_Speed(W);

```

```

end Write_Speed;
      or
        accept Finish ; exit Main_Loop;
      end select;
    end loop Main_Loop;
  end Serialized;

```

6.2 User_Interface package

This package defines two tasks in charge of the user communication. The Operator task allows the user to introduce a command and execute it. These commands may be: defining the robots and their initial positions, defining the execution mode (virtual or real), defining the operation mode (goal oriented or robot chase) , starting or stopping the robot system, defining the arena (a file with the arena and obstacles is provided), etc.

The Monitor task presents a graphical interface where it is shown how the robot(s) follow their goals. This periodic task updates the robot position and traces the path followed by the robots (figure 4 shows a version of this interface using a Java Browser). It must also display the map representing the environment (in real mode this representation is not available, therefore it would be displayed an environment without obstacles), and the position of the robot in each instant.

6.3 Scenario_map package

This package manages the information related to the environment (static) and the current positions of the robots. The specification of the package is provided in the net lines.

```

with Global_Types; use Global_Types;
package Scenario is

  protected type Scenario_Map is
    procedure Define_Scenario(F: in File_Type);
    function Get_All_Robots_Position return All_Positions;
    function Get_Robot_Position(R: in Robot_Id)
      return Position;
    procedure Set_Robot_Position(R: in Robot_Id;
                                Robot_Pos: in Position);
  private
    Arena : Map;
    Current_Robot_Positions : All_Positions;
  end Scenario_Map;
end Scenario;

```

6.4 Java implementation

To implement communication between nodes in a distributed environment, RMI is used. RMI is a Remote Method Invocation system that assumes an homogeneous environment of the Java Virtual Machine. It provides a distributed object model similar to Java. RMI permits to invoke remote interfaces methods, passing a reference to a remote object in an invocation, and it supports callbacks from servers to applets.

A short comment of every object is made in the next subsections.

6.4.1 User interface

This package could be implemented in two different ways: as a frame if we want to work with a console application or as an applet if we want to work with a browser. The following code show the two options :

```
package User_Interface;      package User_Interface;
import java.awt.*;           import java.awt.*;
import java.rmi.*;           import java.rmi.*;

public class                 import java.applet.*;
UserInterfaceRMI
extends Frame;
{
    // Methods
    .....
}

public class
UserInterfaceRMI
extends Applet;
{
    // Methods
    .....
}
```

6.4.2 Robot controller

Robot_Controller includes Robot_Status and Robot_Interface packages. It is implemented as a extends UnicastRemoteObject to allow RMI communication.

```
package Robots_Controller;
import java.rmi.*;
public interface Robot_Controller
    extends Remote;
{
    // Methods
    void Define_Robot (id: Robot_Id);
    void Define_Execution_Mode( m : Execution_Mode);
    ...
}
```

Robot_Status package have the same interface as Ada95 version, otherwise Robot_Interface change in the way that it has two objects implementation (one for the virtual mode and other for the real mode) and one interface.

```
package Robots_Interface;
public interface Robot_Interface
{
    // Methods
    int Get_Next_Sensor(int Id_Sens);
    int Get_Left_Encoder(void);
    .....
}

package Robots_Interface;      package Robots_Interface;
public class Real_Robot         public class Virtual_Robot
implements Robot_Interface;     implements Robot_Interface;
{
    .....
}

package Robots_Interface;      package Robots_Interface;
public class Real_Robot         public class Virtual_Robot
implements Robot_Interface;     implements Robot_Interface;
{
    .....
}

package Robots_Interface;      package Robots_Interface;
public class Real_Robot         public class Virtual_Robot
implements Robot_Interface;     implements Robot_Interface;
{
    .....
}

package Robots_Interface;      package Robots_Interface;
public class Real_Robot         public class Virtual_Robot
implements Robot_Interface;     implements Robot_Interface;
{
    .....
}
```

7. DISTRIBUTION

According to the Distributed Systems Annex[5], the distributed implementation of the robot benchmark requires to identify a number of Ada95 partitions that can execute independently on different nodes. These partitions are Remote Call Interface (RCI) (pragma Remote_Call_Interface(...)) units that provide a set of

procedures that can be called remotely from other partitions. Two basic partitions have been identified:

- The Robot_Controller partition: it provides a high level interface for commanding the robot. It basically provides methods for defining the robot operation, that consists of defining its mode and its targets. This partition groups the Sensor task, the Control task, the Supervisor task, the Robot_Interface package and the Robot_Status package.
- The Console partition provides a graphical interface for displaying the robots operation and for robot commanding. It consists of the Monitoring and the Operator tasks, and Scenario_Map package. This partition has been also implemented in Java using a web browser. In this case, the console is a typical application of Applets (figure 5).

In the obstacle avoiding application there are only two partitions: a Robot_Controller partition that acts as a server and the Console partition that is the client. In the robot chase applications there are as many Robot_Controller partitions as physical robots plus a Console partition. This application has an added problem over the previous one: how the robots learn about the position of the other robots. To solve this problem it is necessary to implement a robot group abstraction. Below a robot group communication is explained.

7.1 Group communication

In robot chase applications, the position of a leader robot must be known to the group of follower robots at each sample period. Group communication then becomes an important issue. This aspect can be addressed in two main ways:

- client-server communication: the followers (clients) invoke a leader method (server) to get its current position.
- broadcaster-listener communication: a group manager abstraction is used for this purpose. Followers join a group when they become the followers of a given leader and the group manager broadcasts the leader position upon change. If a broadcast facility is not available, it can be replaced by a iteration loop, including all followers, invoking a method to inform the follower about the leader position.

The Java implementation uses the second approach while the Ada95 implementation is based on the first one. It uses a Robot_Position object where all robots record their positions periodically and where other robots can read them. This alternative has the drawbacks of managing remote communication inefficiently and, most importantly, it yields cumulative errors in robots positions due to robots sliding on the arena. To overcome this problems it has been planned to replace this solution by a new one based on a camera in

zenithal position over the arena that will broadcast the robots position to the group of robots. In this case, the digital

camera would be modeled as a new partition.

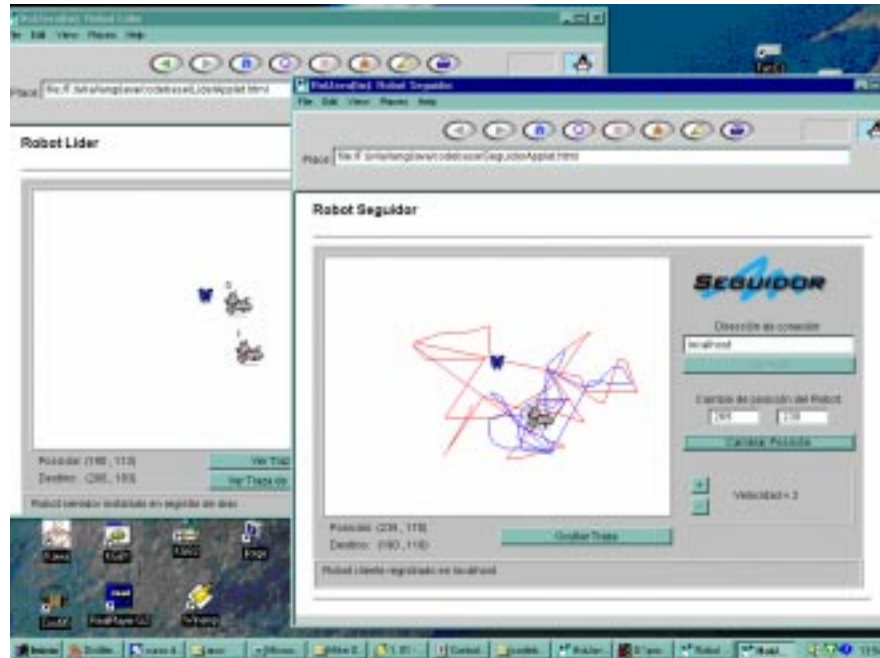


Figure 5. Application snapshot

8. STUDENTS WORK

The robots application provides a playground for experimenting with real-time distributed control systems but, up to now, it has not been described the exercises and projects that the students accomplish on this platform. This exercises depend on the subject they are attending.

In Real-Time Systems the focus is on the structuring of the application as a set of periodic tasks, on simulating the physical process and controlling it and, finally, replacing. All the components related to the user interface, such as the operator and display tasks, are provided in advance. On this basis, the students develop several experiences. The goal of the first experience is modelling a physical process. Concerning this goal the students begin by implementing a robot simulator (Robot_Interface package). In the next experiences, the emphasis is put on the control of the robot and periodic tasks. This way they implement the Control task first, and the Supervisor task next. The last set of experiences are oriented towards replacing the simulated robots by real ones (Khepera robots) and system integration. Also, the students have to analyze the system (task measurement) and priority assignation to the task. A schedulability tool is used to perform the schedulability analysis.

In Distributed Systems the focus is on structuring the system as a set of remote objects (partitions), developing the needed components for remote communication (stubs), configuring the system and developing the user interface using web

browsers. For these experiences a non-distributed implementation of the Robot_Controller and Robot_Interface are provided in advance. For this part, the students mainly use the Java language. The experiences consist of adapting the server components (done in advance) for remote communication and entirely developing the client part (Console module). There are three experiences using different communication mechanisms: sockets, Java-RMI and CORBA (on Java). The client part in a progressive way: in the first three experiences they build a “command line” application (no graphical interface). In the two last ones they develop a graphical applet using a visual development environment.

In this paper we have examined the experiences using Ada in the Real-time Systems and Distributed Control Systems laboratory of the Control Engineering at the Politechnical University of Valencia (Spain). The paper has detailed the application based on virtual or real robots and the design supplied to the students to work on it. The experience can be considered as highly satisfactory because of the students start from a preliminary design and implementation and can develop more and more functionality.

9. CONCLUSIONS

The availability of two complementary environments (simulated and real) offers important advantages. Firstly, the students demand the use of real processes to experiment with it. But, on the other hand, they appreciate very much the simulated environment because of the virtual operation

allows to experiment in new algorithms to avoid obstacles, to simulate the robot, etc., and most of these modifications are faster in a simulated experience than a real with robots.

The same experience has been developed using a pilot plant of residual water. In this case, the process and controller are simpler than the robot, for this reason it is used as first experiment during the courses.

Now, we are working in the execution of this experiment under the RT-Linux [6] operating system. In order to develop embedded system, we are working to in the execution of Ada programs under the JTK [7], a library which provides simple tasking support, loosely based on POSIX threads. It has been designed to provide low level tasking support for GNAT, it is completely written in Ada, and it is usable on top of a traditional operating system (providing user-level threads) as well as on bare machine.

10. REFERENCES

- [1] A. Crespo, J. Vila, F. Blanes, I. Ripoll. Real-Time Education in a Control Engineering Curriculum. Real-Time Systems Education. IEEE Computer Society Press. 1998
- [2] W.A. Halang. , J. Zalewski. A Model for Real-Time Systems Curriculum.. Real-Time Systems Education. IEEE Computer Society Press. 1996, pp: 39-48.
- [3] A. Burns and A. Wellings, Real-Time Systems and their programming languages(2nd Edn), Addison-Wesley, 1996.
- [4] Khepera User Manual. <http://www.k-team.com/>
- [5] Braitenberg, Valentino, Vehicles: Experiments in Synthetic Psychology, MIT Press, 1987.
- [6] M. Barabanov, V. Yodaiken. Real-Time Linux. New Mexico Institute of Mining and Technology. 1997.
- [7] Juan A. de la Puente , José F. Ruiz, and Jesús M. González-Barahona. Real-Time Programming with GNAT: Specialised Kernels versus POSIX Threads. 9th International Real-Time Ada Workshop, IRTAW'99.

