CO-OPERATIVE ROBOTICS SIMULATOR - ROBOT SIMULATOR

by

VENKATA PRASHANT RAPAKA

B. E., Osmania University, India, 2002

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2004

Approved by:

Major Professor
Scott A. Deloach, Ph.D.

ABSTRACT

In this report, the goal to develop a component in the Cooperative Robotics Simulator, the design and implementation to that effect is documented. The Cooperative Robotics Simulator can be used to perform simulations of many (one or more) heterogeneous types of robots all working within a single, virtual environment. A simulator is primarily used to replace a static workspace with a virtual one which is more flexible and possibly dynamic. The Cooperative Robotic Simulator provides a framework to create various topographies with different robot species each with a variety of features. A cooperative robot can perform actions based on information gathered from the workspace and other robots. These actions of the collaborating robots can be tailored to solve a bigger problem.

Specifically this report documents the design and implementation of the robot objects that emulate the behavior of real robots in the Environment Simulator. This includes the emulation of various robots and their peripherals by the robot objects and the effects of the actions of robots in the environment by the hardware simulator. The peripherals include periodic sensors like sonar sensors and non-periodic sensors like bump and heat sensors and actuators like wheels. I also wrote the control code that governs the behavior of each robot. The robots objects run the control code containing commands which are broken by the hardware simulator into synchronized time slices of execution in the workspace executed by the Environment module. All these features have been tested by simulating swarm algorithms based on Ant Colony Optimization. An analysis of the results and conclusions drawn from these test results are also presented.

KEYWORDS: Cooperative Robotic Simulator, Ant Colony Optimization, Environment

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDEMENT

This project is a significant accomplishment of my academic life and none of this would have been possible without the presence of some special individuals who have taken their rightful place in my life.

I owe my success in this project to my advisor Dr. Scott Deloach whose constant direction and advice have helped me understand and evaluate my work. I would like to thank Dr. David Gustafson and Dr. William Hankley for serving on my committee. I am grateful to my colleagues in the Computer Science Department especially Scott Harmon, for helping me see with a different perspective and solve some of the problems I have faced during the course of this project.

I am deeply indebted to my parents who have always been by my side and to my brother without whose constant support and encouragement I would have given up long back.

Chapter 1 Introduction

1.1 Introduction

Simulation allows researchers, designers and users to construct robots and environments for a fraction of the cost and time of real systems. They differ significantly from traditional CAD tools in that they allow study of geometries, kinematics, dynamics and motion planning. Most existing dynamic simulators are for either specific types of environments or for simulating motion of specific type of robots. Therefore, their applications to more complex structures with varied scenarios comprising of different types of robots and environments have been very limited. These limitations force the programmer to go through substantial reimplementation when the simulators need to accommodate simulations of very general environments. Also, it is almost impossible to extend their functions to be able to perform simulations of different paradigms due to their lack of flexibility in its design principle for accommodating different types of simulations.

The Cooperative Robotics Simulator attempts to achieve this flexibility at the cost of loose coupling in its architecture. It provides abundant features to implement different topologies in which the behavior of cooperative robots can be simulated and observed. This improves testing various scenarios as now complex virtual environments are easy to build and the simulation can be run at a desired speed.

1.2 Cooperative Robotics Simulator Overview

The Cooperative Robotics Simulator is a platform to simulate a workspace. It emulates a class of robots specifically the Scout in the simulated workspace. The simulation is influenced by different external users providing various inputs. It provides tools to create virtual environments that are supplied to the Simulator as input. It also contains tools to observe the simulation live or as a recorded replay. The robots' behavior in the simulation is controlled by running control code consisting of commands supported by the Scout. An overview of the various modules in the application is shown in Figure 1.

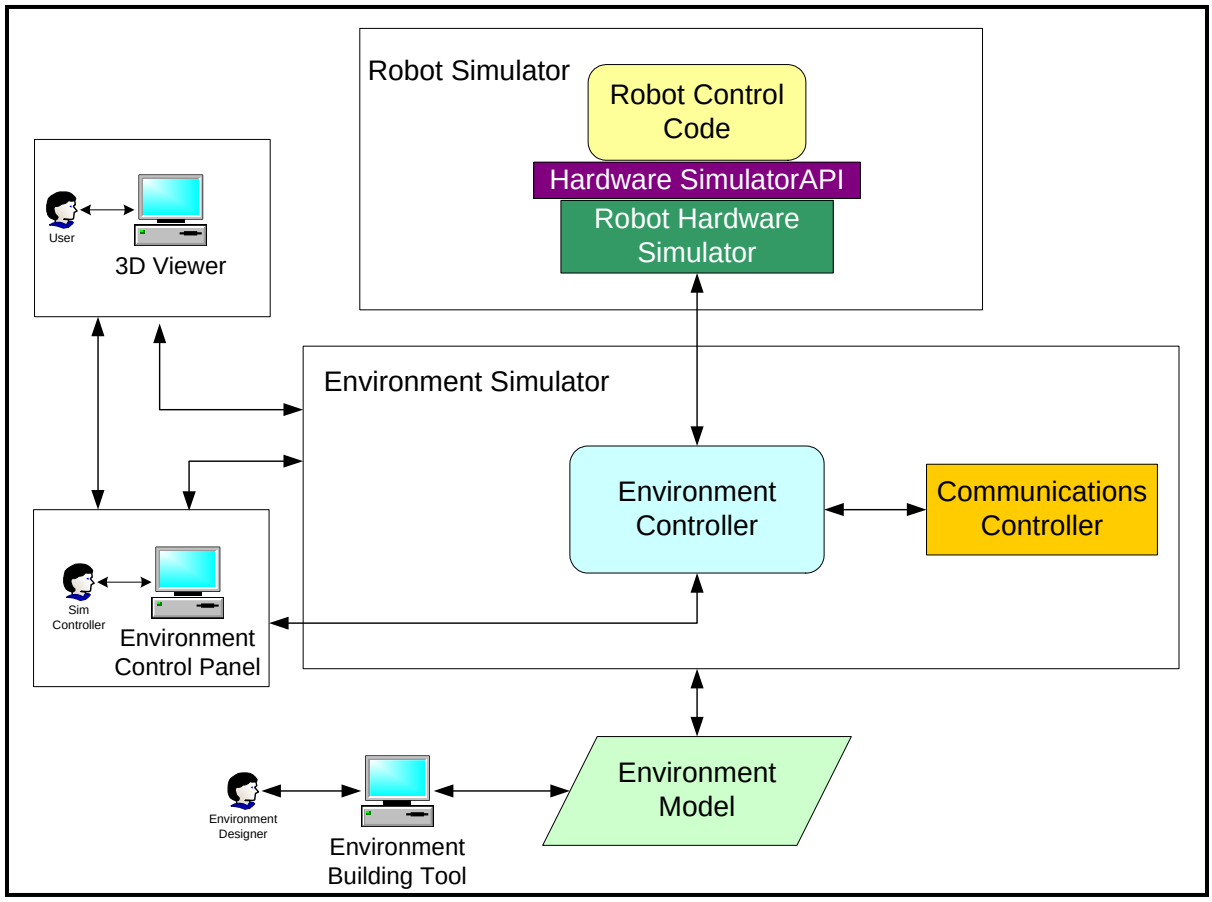

Figure 1: Co-operative Robotics Simulator Overview

1.3 Cooperative Robotics Simulator Components

The simulator contains many interacting components to replace various elements of a real workspace. The main components are Environment Simulator, Communication module, Environment Control Panel. CRS Viewer and Environment Model Building Tool provide auxiliary services. In this project I have designed and implemented the Robot Simulator and integrated it with other components in the Cooperative Robotics Simulator. Each of these components and their respective role is explained in the following sections.

1.3.1 Robot Simulator

The Robot Simulator consists of three parts: a robot hardware simulator, a robot control program, which will be user supplied, and a robot object. A standard API is defined between the robot control program and the robot hardware simulator. This API allows robot control programs to work with various robot hardware simulators. Robot hardware simulators are defined by the set of standard sensors and actuators that they contain. While in real life, the sensors and actuators reside on the robot hardware; they are actually under control of the environment-based robots to allow degradation.

1.3.1.1 Robot Hardware Simulator

The Robot Hardware Simulator is interfaced to the environment via requests for data from its sensors or requests for action from its actuators. The robot-objects control the individual sensors, based on robot hardware simulator requests, and providing the appropriate data to the sensors for feedback to the robot hardware simulator.

1.3.1.2 Sensors

There are separate sensor models for each hardware sensor available to a robot hardware simulator. Sensor models will include sonar, bump sensors, infrared sensors, various cameras, compasses, GPS sensors, etc. The sensors are coded to take data from the environment model and return that data as interpreted by the sensor. For instance, a sonar sensor model would take as inputs its current location, orientation, and environment model data and outputs a value related to the distance of the closest object in its view.

1.3.1.3 Actuators (Effectors)

There are separate actuator models for each hardware actuator available on a real robot. Possible actuators are movement actuators, motors, grippers, arms, etc. The actuator models will be implemented to take actuator requests from robot hardware simulators and, based on environment model data and degradation parameters, provide the actual effect on the environment. This output will be used to update the environment model. Again, the output of the actuator models will be probabilistic and will account for degradation, under control of the environment.

Other components in the project are introduced in the following sections.

1.3.2 CRS (Cooperative Robotics Simulator) Viewer

The CRS Viewer is a visualization tool that displays a three dimensional view of the simulation to the user. The viewer is capable of moving the view as in zooming in and out of a location and panning through the panorama. The 3-D viewer is also customizable in terms

of camera angles, light sources and infra red vision. The record and replay modes in the viewer aid in obtaining a better understanding of the simulations in different perspectives.

1.3.3 Environment Simulator

The Environment Simulator is the central component in the system. It keeps track of the actual state of the environment, including each robot. It also receives requests from simulated robots to read sensors, initiate actuators, and to send and receive communications among robots in the environment. The environment will provide sensor data to simulated robots by passing requests to the appropriate environment-based robot object, which uses sensor models that transform environment model information into the appropriate sensor output data. The environment updates the environment model by taking requests from simulated robots to perform actions on the environment via environment robot actuators. Again, the environment simulator determines the effect on the environment model using the output of actuator models.

The environment simulator initializes and maintains the entire simulation. It reads the environment model from an initialization file. Robot simulators and viewers then register with the environment simulator.

1.3.4 Communication

All communication will be in the form of messages sent to and from robots using capabilities implemented in various communications models. These messages include information regarding the sender, receiver, type of message being sent and the actual content of the message. Communication is handled as part of the environment communication model to allow control over delivery of messages in synchronization with timestep and other

factors like delay and degradation. The types of communication possible will depend on the communication model used like broadcast, multicast, etc.

1.3.5 Environment Control Panel

The environment control panel is a standalone system that connects to the environment simulator to monitor and control the current simulation. Specifically, the environment control panel is capable of viewing 2-D and 3-D representations of the environment from various angles including an overhead, or God's eye view. The environment control panel will also be able to monitor all communications and shutdown all or some communications. The control panel will also allow the user to select and monitor/change the status of individual robots within the environment. Specifically, the user should be able to degrade individual sensors or actuators on the individual robots.

1.3.6 Environment Model Building Tool

The Environment Model Building Tool allows the developer to create new objects by composing simple object types. The output of the tool would be an environment model in the appropriate file format. The environment model is initialized from an environment model file. The environment model will contain a description of the environment surface, including inclines, as well as a description of objects within the environment.

More information regarding the Cooperative Robotics Simulator project, project members and current status can be obtained from the project website [6].

Chapter 2 Scout

2.1 Introduction

In the project the Robot Simulator simulates the Scout class of robots in the environment. The Scout is an integrated mobile robot system developed by Nomadic Technologies [1]. It is capable of ultrasonic, tactile and odometry sensing. It is a three wheeled robot with two independent motors. The Nomad sensorial input consists of an array of fifteen sonars and eight bumpers, distributed in its cylindrical body. It uses a special multiprocessor low-level control system that controls the sensing, motion, and communications. At a high level, the Scout is controlled either by a laptop mounted on top or a remote workstation communicating view radio modem. Alternatively, the Scout is controlled via an on-board PC computer. Currently, the host computer software must run under Linux.

2.2 Basic functions for the Scout

A typical control code running on the Scout has the following steps

- Connect the computer to the robot
- Configure the sensors
- Perform desired task
    - The robot reads sensors
    - Perform appropriate actions
- Disconnect from the robot.

The Scout robots can be controlled in many ways like using a joystick, a radio controller or code running either on an onboard computer or on a laptop connected via a serial port. Although the first two are valid means to control the robot, the last two ways are important to the simulation. The control code is a C program consisting of a bunch of commands that are supported by the Scout robot to control and command various peripherals. The code is executed sequentially. Information about the current state of the robot, its configuration and the readings of the sensors can be obtained by an application program through a global array, called the State vector. As the commands in the control code are executed the array is modified at the respective location. Symbolic Constants like STATE_BUMPER, STATE_SONAR_0, STATE_SONAR_1, etc are used as array indices to obtain the current value of the attributes they represent in the state array.

The Scout is code compatible with the Nomad 200 class robots. In order to maintain the Application-Programmer Interface (API) between the Nomad Scout and the Nomad 200 (An older version of the robot), some functions have extra unused parameters. This is because the Scout has one fewer degree of freedom in its motion system than the Nomad 200. The N200 used a synchro-drive system, with one axis for translation, one axis for steering, and one axis for a turret containing the sonar sensors as well as other sensor packages. The Scout employs a differential drive system, in which the user controls the right and left wheels independently. The extra unused parameters in the motion functions should be passed as zero. All the commands supported by the Scout are detailed in [2]. Some of these commands used for each of the steps presented at the beginning of this section are listed in Table 1.

Scout Robot Commands
Communication Commands connect_robot . . . . connects to a robot disconnect_robot . . . . closes connection with a robot conf_tm . . . . sets the timeout period of the robot real_robot . . . . . .switches to real robot mode simulated_robot . . . . switches to simulated robot mode
Motion Commands pr . . . . moves the motors of the robot by a distance vm . . . . . moves the robot at given velocities mv . . . . . moves the three axes of the robot independently st . . . . . stops the robot's motors ws . . . . waits for the stop of the robot's motors
Motion Parameters Setting Commands dp . . . . .defines the position of the robot ac . . . . .sets the robot's accelerations sp . . . . .sets the robot's speeds
Sensor Commands conf_sn . . . . . .configures the sonar sensor system get_sn. . . . . .gets the sonar data of the robot get_bp . . . . . .gets the bumper data of the robot

Table 1: List of commands supported by Scout robots

2.2.1 Host computer communication

int connect_robot (int id, int model, char *device, int v) connects the host computer to the robot identified by id, using port device; if device is a serial port then v is the communication speed in bauds. If the value device is a tcp port, then v is a port number. The parameter model must be the constant MODEL_SCOUT for the Scout robot. For a typical connection using serial port COM1, a call to this function looks like the following:

```
connect_robot (1, MODEL_SCOUT, "/dev/ttyS0", 38400);
```

int disconnect_robot (int id) disconnects the host from the robot identified by id.

For example, to disconnect from the robot from the previous example, write:

```
disconnect_robot (1);
```

2.2.2 Moving the robot

vm (int r_wheel, int l_wheel, int dummy) sets the speeds of the right and left wheels to r_wheel and l_wheel, respectively. Speeds are given in 0.1 in/sec. Variable dummy is not used, but something must be passed to it.

For example, to make the robot move forward at 10 in/sec, write:

```
vm(100, 100, 0);
```

st () stops the robot. When this command is given, the robot starts decelerating, but the program continues to execute. If the next command should be executed when the robot has actually been stopped, a wait command (ws) must be given after the stop command.

For example:

```
st ();                  /* Tells the robot to stop */
ws (1, 1, 0, 255);      /* Wait until both wheels stop */
```

ac (int r_wheel, int l_wheel, int dummy) sets acceleration of left and right wheels. Accelerations are given in 0.1 in/sec^2. To set both wheels to accelerate 20 in/sec^2, type:

```
ac (200, 200, 0);       /* ac= 20 in/sec^2 */
```

pr (int r_wheel, int l_wheel, int dummy) attempts to move the wheels to positions r_wheel, l_wheel relative to their current position. Positions are given in 0.1 inches. A ws command should be given to allow time for the robot to complete the movement before the next program line is executed. For example, to move the robot 20 inches forward, write:

```
pr (200, 200, 0);
ws (1, 1, 0, 255);
```

2.2.3 Differential drive macros

For some applications, it may be useful to give movements commands in terms of how much we want the robot to translate, and how much we want it to rotate. Translations are given in 0.1 in., steers in 0.1 degrees. The following methods achieve this:

scout_vm (int trans, int steer) to give the translational and rotational speeds

scout_pr (int trans, int steer) to give the relative translation and rotation from the current robot position

For example:

```
scout_vm (0,1800);      /* Turns the robot around by 180° */
scout_pr (0,900);       /* Turns the robot to its right by 90° */
```

2.2.4 Configuring sensors

int conf_sn (int rate, int order[]) fires a sonar every rate*4 ms. Sonars are fired in the order specified in the array order. Variable rate is in the interval from 0 to 255. If not all of the sonars are used, order should end with a 255.

For example:

```
int s_order[6] = {0, 1, 4, 12, 15, 255};
conf_sn (15, s_order);
```

2.2.5 Reading Sensors

get_sn ( ) updates sonar readings in the State array. Sonar readings are stored at indexes STATE_SONAR_0… STATE_SONAR_15 and are given in 0.1 inches.

For example, the following code gets the latest reading of sonar number 1:

```
get_sn ();
dist = State[STATE_SONAR_1];
```

get_bp ( ) obtains a fresh reading from the bumper array of the robot. This function updates array State at index STATE_BUMPER. When State[STATE_BUMPER] is different from zero, it means a bumper has been pressed. The n^{th} bit of State[STATE_BUMPER] corresponds to the n^{th} bumper.

For example:

```
get_bp();        /* Gets the current state of the bumpers in to the state array */
```

Chapter 3 Robot Objects

3.1 Architecture

The Robot-Object component emulates one among many agent architectures with facilities for intelligent multi-agent communication, navigation, localization and calibration. Each type of robot supports a collection of commands and possesses a collection of peripherals. Each robot in turn will comprise of attributes describing the number and type of peripherals and their relative position to the robot. The Robot Object encapsulates the features and actions of a specific robot in the environment. The generic implementation of the Robot Objects allows simulating robots with different collection of sensors and actuators. The Robot-Objects are capable of emulating the Scout class of robots.

The Robot Objects are implemented as a hierarchical framework with each class in the framework defining one of the robot tasks described in the previous chapter. At the top of the hierarchy is the interface CoOpRobot that defines the common services required by any type of robot. The Robot-Objects emulating different types of Robots are encapsulated in their respective classes that implement the CoOpRobot Interface. An overview of the associations existing among classes implementing the robot-objects is shown in Figure 2.

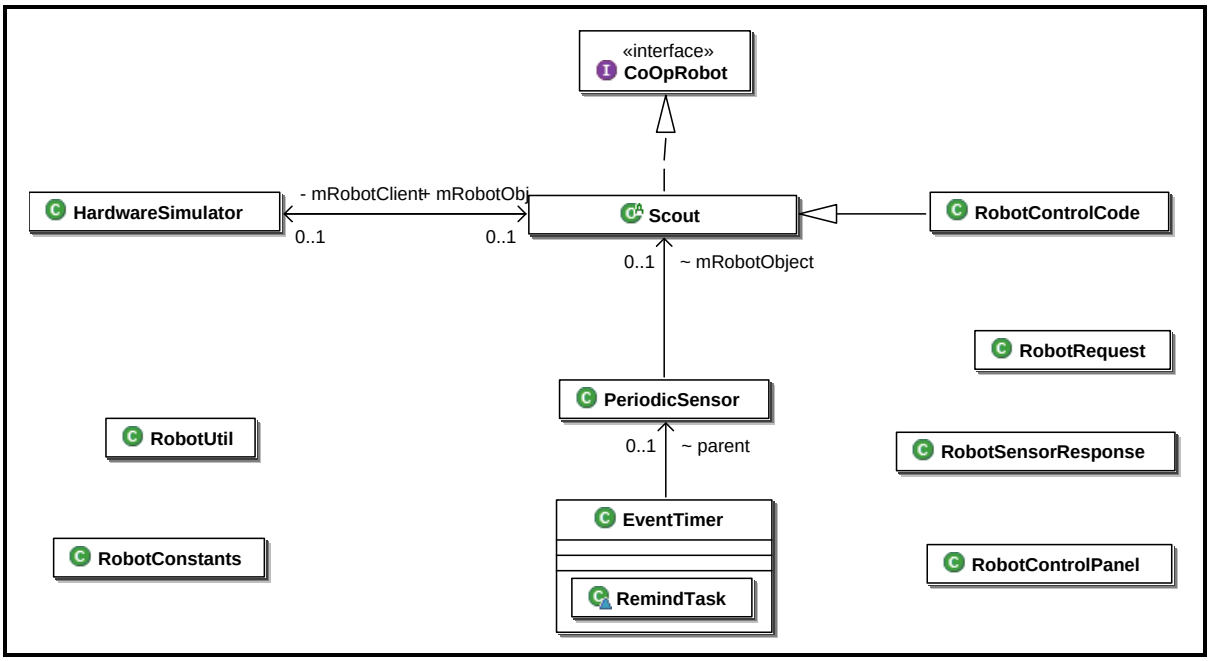

Figure 2: Architecture of the Robot Simulator

3.2 Overview of robot package

At the top of the class hierarchy in the robot package is the CoOpRobot interface that defines the operations common to all robots like communication with the Environment. Each class of robot implements the CoOpRobot interface to add the implementation of their own respective commands. Each robot contains an instance of hardware simulator to emulate its sensors and peripherals. The periodic sensors on the Scout are emulated by the PeriodicSensor class. The PeriodicSensor class creates an EventTimer class which generates events at the frequency the periodic sensor being emulated is to be used. The PeriodicSensor class creates sensor requests on receiving events from the EventTimer class. The RobotRequest and RobotSensorResponse classes encapsulate the information in a request sent to the Environment and the corresponding response returned. The RobotControlPanel class is the platform to start and manage robot objects and RobotControlCode class contains the commands to be run on the robots during simulation.

The robot package also contains some auxiliary classes to separate fixed constants and user-defined library methods from the Robot Simulator implementation. The RobotUtil class provides generic services to the whole application like controlled message output, generating a representation of the state of a robot at any instance in simulation etc. The RobotConstants class contains the fixed constants in the application like location of execution, conversion factors and attributes of the class of robot being simulated. The associations and inheritance explained so far in the robot package are depicted in Figure 3.

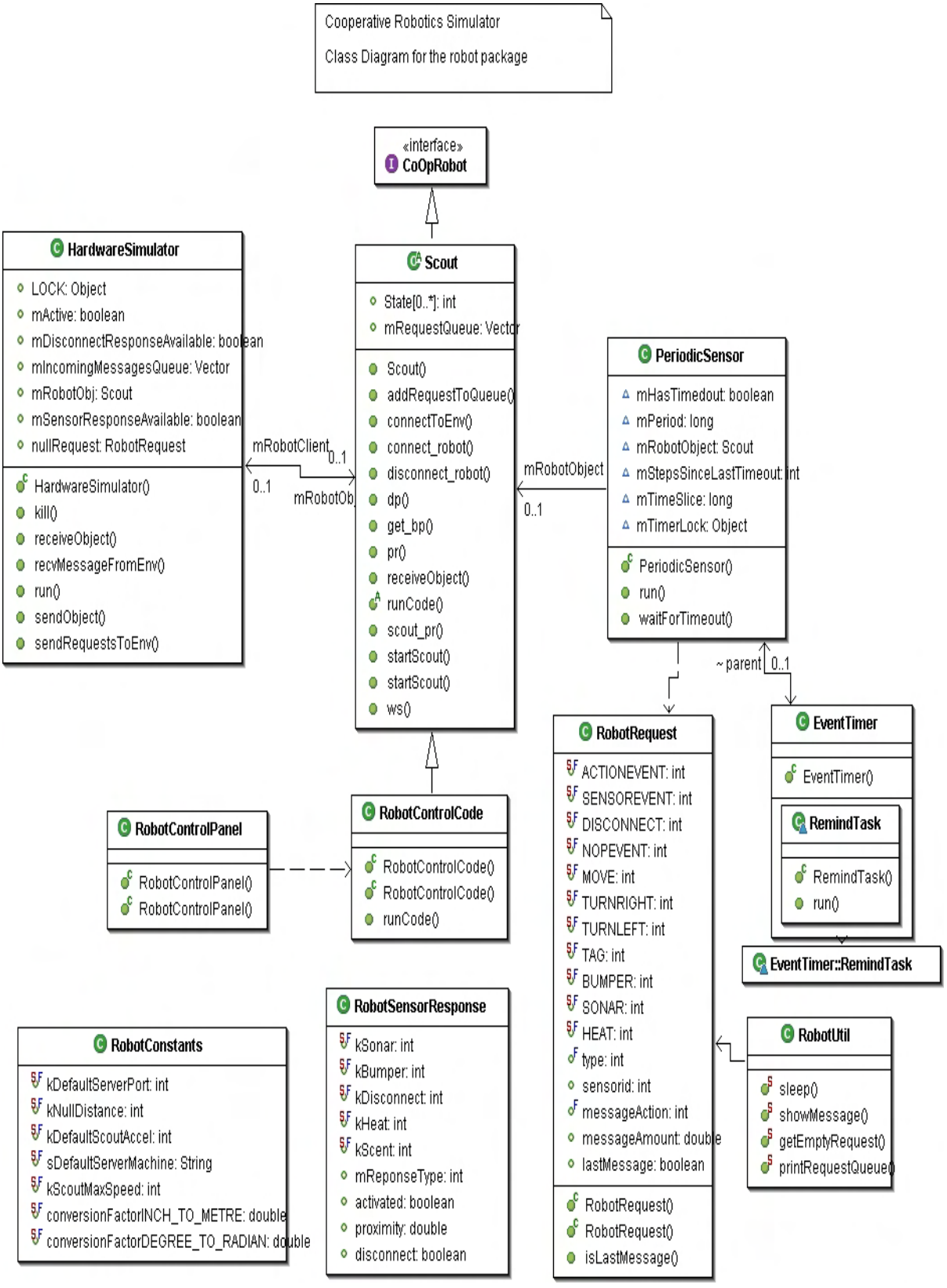

Figure 3: Class diagram of the robot package

3.3 The Scout Class

The Scout class encapsulates the class of Scout robots presented in chapter 2. Every Scout has a set of values representing the state of various sensors, actuators of the robot and current physical attributes like speed, orientation and position. The Scout class contains an array State to store these values. The attributes are stored at predefined locations in the State array as mentioned in the Scout user Manual [1].

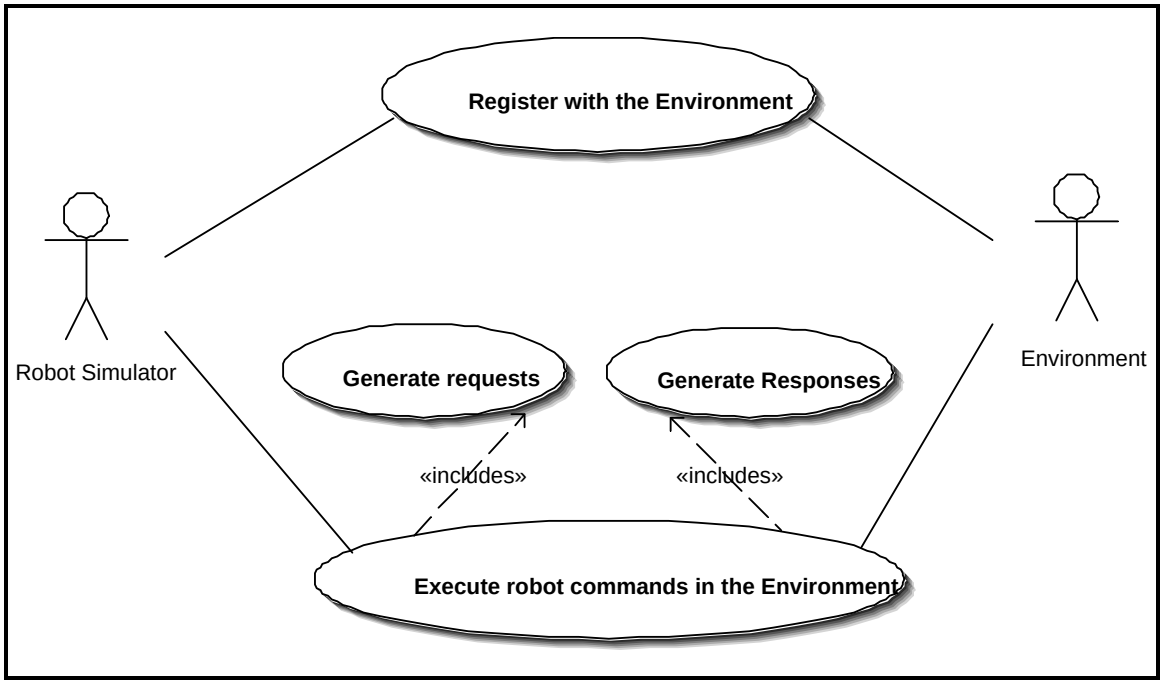

Figure 4: Use case diagram of the interaction between Robot Simulator and Environment

The Scouts support a set of commands to start the robot and use the sensors to control its movement based on the information gained about the surroundings. The Scout class in the robot package implements a subset of the commands supported by the Scout robot. The commands implemented by the Scout Class can be classified based on their use.

- Communication Commands
    - void connect_robot (int id, int model, String device, int v)
    - void disconnect_robot (int id)
- Motion Commands
    - void scout_pr (int trans, int steer)
    - void pr (int r_wheel, int l_wheel, int dummy)
- Motion Parameters Setting Commands
    - int dp (int x, int y)
    - void vm (double speed)
    - void st ()
    - void ws (int x, int y, int z, int dummy)
- Sensor Commands
    - int conf_sn (int rate, int order[])
    - void get_bp ( )
    - void heatSensorCheck ()
    - void sonarSesnorCheck ()

The various fields and methods in the Scout class are depicted in Figure 5.

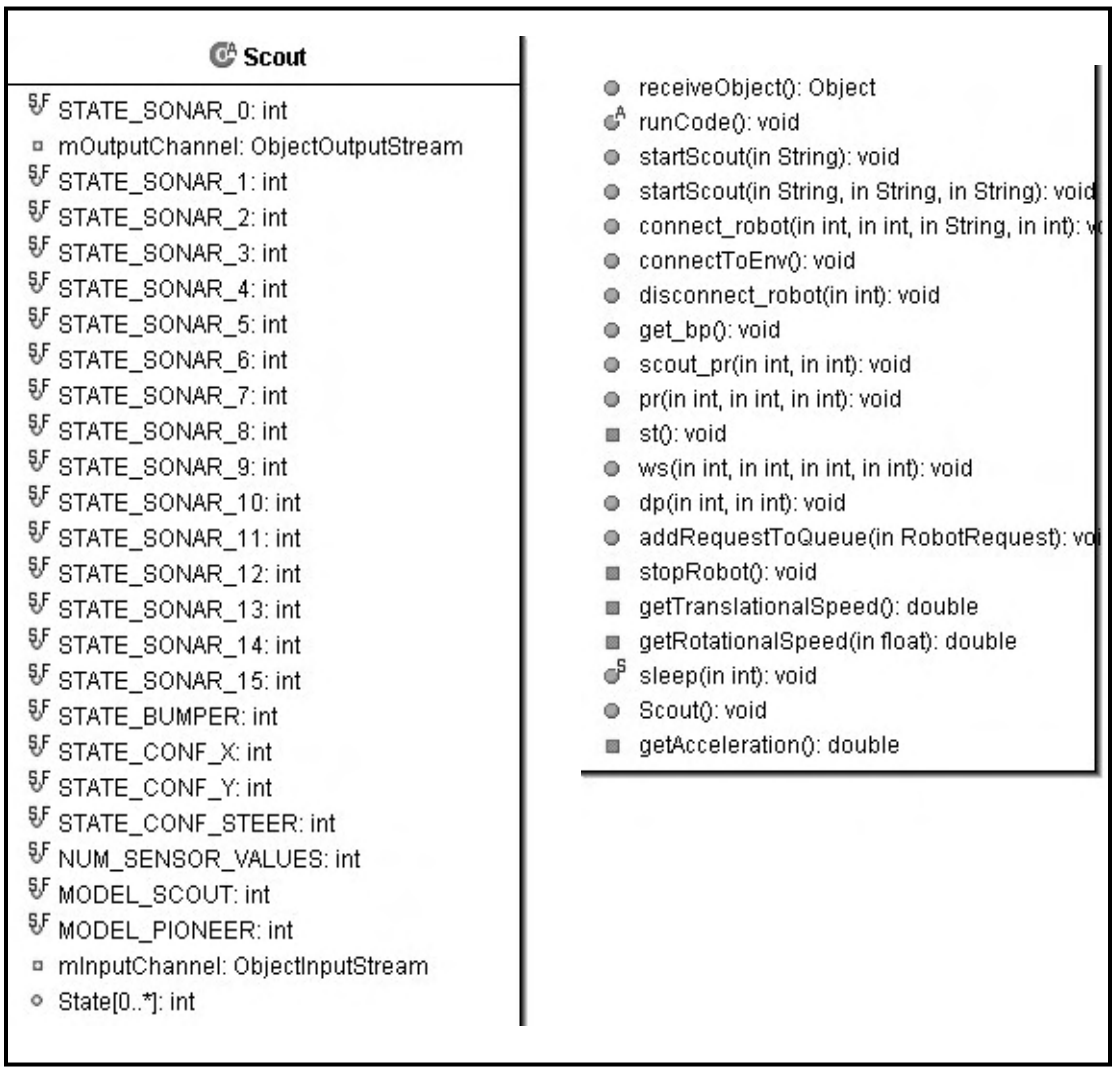

Figure 5: Class Diagram of the Scout class

3.4 Control Code class

The ControlCode class contains the code to be run on the robot during simulation. Its capability to run the robot commands comes from inheriting from the class encapsulating the attributes and methods of the type of robot being simulated. A part of a control code program is shown as an example in Table 2.

```
/*
* This program will connect to a scout robot,
*  use commands for configuring and controlling movement,
*  and sensory information
*/


/* Connection */
int ROBOT_ID = 1;
int SERV_TCP_PORT = 7019;
String SERVER_MACHINE_NAME = "procyon.cis.ksu.edu";

if (!connect_robot(ROBOT_ID, SERV_TCP_PORT, SERVER_MACHINE_NAME)  {
        printf("Connexion to robot failed\n");
        return(1);
}

        dp(100,100);   /* Set the location of the robot as 100,100 */
        vm(10);/* Set the translational speed of the robot to 1 inch/sec */

        heatSensorCheck();      /* Update the heat sensor reading in the State array */
        if (State[STATE_HEAT] == 1)         {

     /* Robots turns until the it is oriented towards any object sensed by the sonar */
                        sonarSensorCheck();
                        while(State[STATE_SONAR_1] == -1)         {
                                scout_pr(1, 100);
                                ws(1, 1, 0, 255);

                                sonarSensorCheck();
                        }

                    /* Move towards the object sensed by the sonar*/
                        while(State[STATE_BUMPER] != 1) {
                                pr(100, 100, 0);     // Move forward 10 inches
                                ws(1, 1, 0, 255);   // Wait for the completion of the move
                                get_bp(); // get bumper readings into State
                        }
        }
disconnect_robot(ROBOT_ID);
return(0);
}
```

Table 2: An excerpt from a typical control code program

3.5 The Hardware Simulator Class

The HardwareSimulator class emulates the various sensors and actuators on a robot. The functionality of the class is depicted in the use-case diagram in Figure 6.

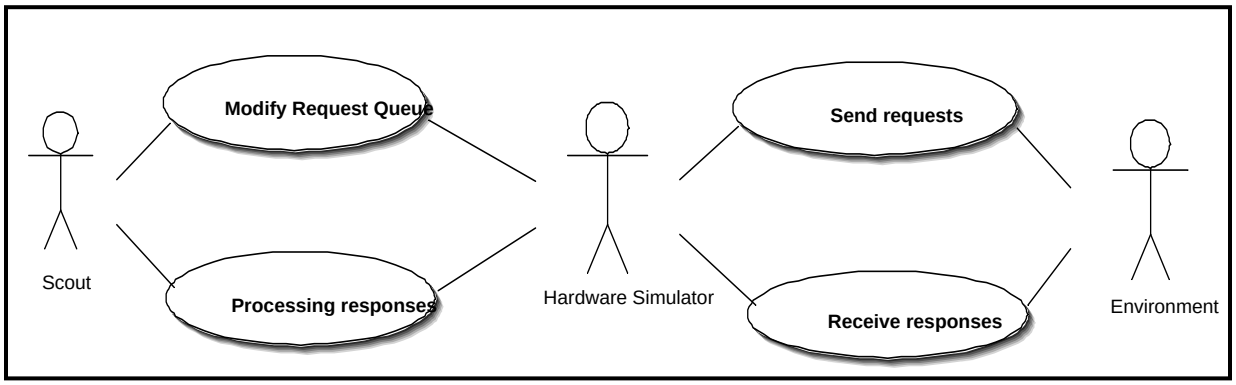

Figure 6: Use case diagram for the interaction between Robot, Hardware simulator and the Enviroment

The important methods in the HardwareSimulator class are the recvMessageFromEnv and sendRequestsToEnv that provide a channel between the Robot Simulator module and the Environment module. Once instantiated by a robot-object the HardwareSimulator busy loops for each timestep from the Environment. When a timestep is received the requests for that timestep are removed from the request queue and sent to the environment. If any sensor request were sent among the request sent for a timestep, the HardwareSimulator waits for a response for each of those requests and modifies the State array in the robot-object from the response. The various fields and methods in the Hardware simulator class are depicted in Figure 7.

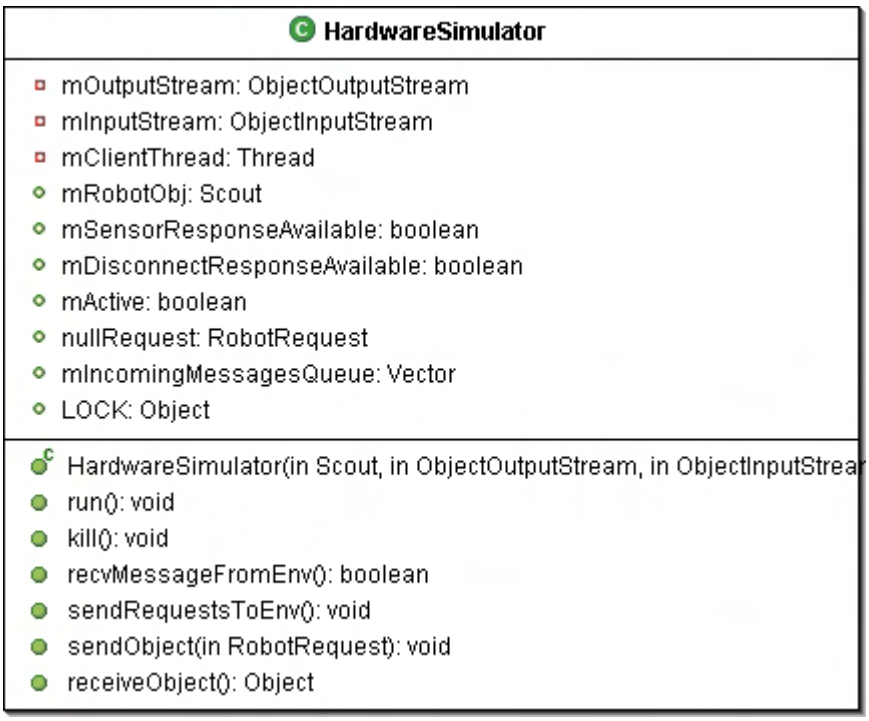

Figure 7: Class Diagram of the Hardware Simulator class

3.6 Communication Protocols

The Robot Objects use two standard protocols in their communication among its classes and with other components in the simulator. The protocol between the Robot-object and their hardware simulator is used to move the commands being executed on the robot to the hardware simulator. The other protocol that is used between the Robot module (Hardware Simulator) and the Environment module transfers the commands to reflect their effects in the environment.

3.6.1 Protocol between the Robot-object and the Hardware-Simulator

The hardware simulator will abstract the execution of those commands in the environment from the robot objects. The robot object generates the requests with a timestep in which they should be serviced that are queued by the Hardware simulator. If robot-object makes a sensor request then it waits till the hardware simulator receives a response.

Robot-Object	Hardware-Simulator
Robot-object with the configured sensors and actuators is created. A hardware simulator is instantiated.	
	The hardware-simulator connects to the environment.
The control code is executed. Requests are sent to hardware simulator.	
	The requests received are added at the end of the request queue.
If a sensor request was made in last timestep Wait for the hardware simulator to intimate that a response is received for the request.	
	If a sensor request is sent Wait for a response from environment to intimate the robot-object.
If a disconnect request is sent Wait for a response. If the response is disconnect Stop the hardware simulator Otherwise Wait for next response	

Table 3 : Protocol between Robot-Object and Hardware Simulator

3.6.2 Protocol between the Environment and the Robot

The robot connects to the environment, receives the timeslice environment executes in each timestep. It then creates requests from control code for each timestep. Whenever it receives the next timestep from the Environment, it sends the requests for that timestep.

An overview of the messages exchanged between the Robot Simulator and the Environment is show in Figure 8.

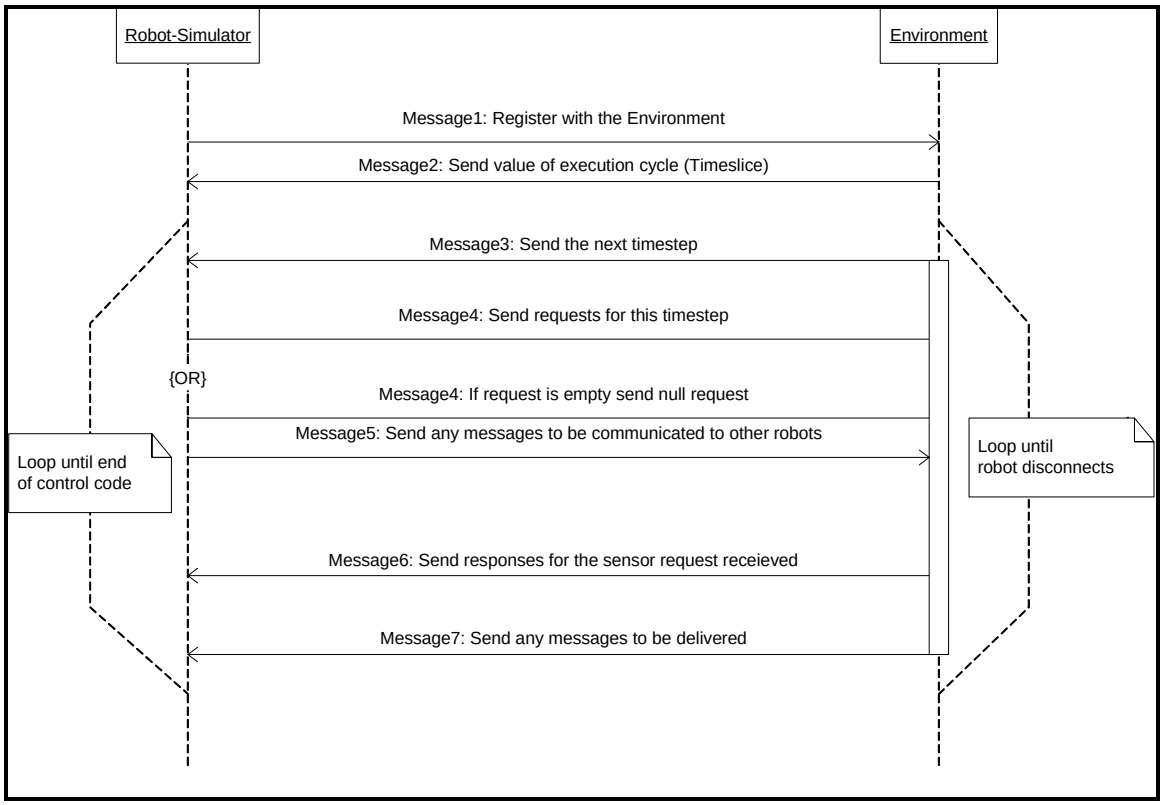

Figure 8: A Sequence diagram of the interaction between the Robot Simulator and the Environment

A more detailed diagram showing the role of the hardware simulator is given in Figure 9.

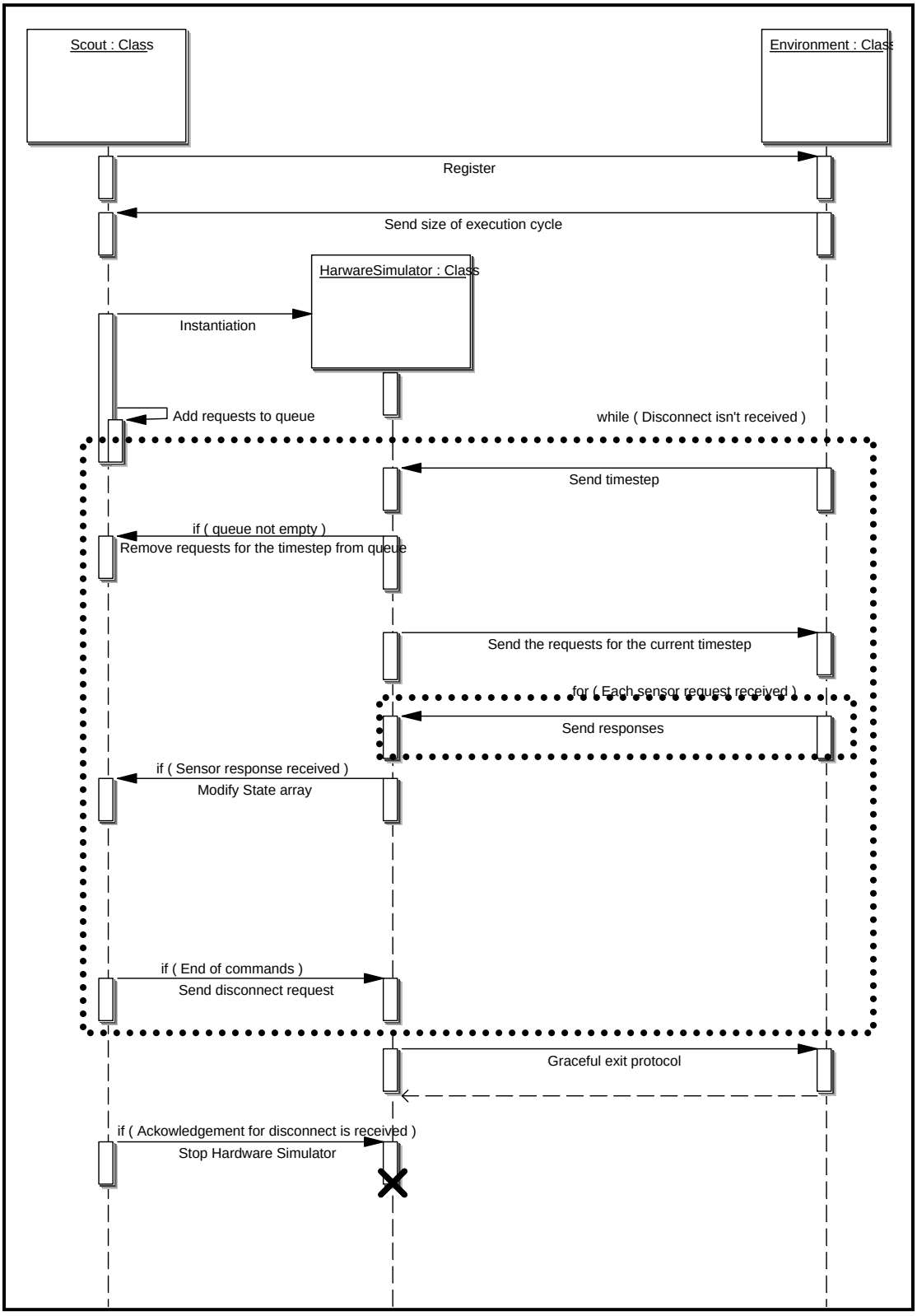

Figure 9: Sequence Diagram for the communication between the Robot Simulator and the Environment

Chapter 4 Sensors

4.1 Introduction

Sensors act as sources of information about the environment and as a form of feedback for the robot by providing a snapshot of simulation. Although a single sensor may not be helpful in creating complex operations, a group of different kinds of sensors can yield sufficient information to a robot to perform significant activities in an environment.

In the Cooperative Robotics Simulator the types of sensors emulated are dictated by those possessed by the class of robot being simulated. The sensor requests are blocking in the sense that the requests for the next timestep aren't sent to the environment until the responses for all the sensor requests made in the current timestep are received by the robot. The sensors are implemented as either as a Periodic sensor or as a Non-periodic sensor.

4.2 Periodic Sensors

4.2.1 Overview

The Periodic sensors gather information about the environment by periodically sensing the surrounding. These sensors are used after preset time intervals and are usually limited in range and accuracy due to the frequency of use.

The Periodic Sensors are implemented as components that generate time-triggered requests. All periodic sensors derive from the parent PeriodicSensor class that contains an

instantiation of an EventTimer to generate interrupts at specific time intervals. At each interrupt a method is called that creates the sensor request to be sent to the Environment.

4.2.2 Sonar Sensor

The Sonar is a tactile directional sensor that provides a quick, simple way to determine approximate distances from obstacles. This information can be used for Navigation, Detection, Collision Avoidance and Obstacle Avoidance. Although the Sonar sensor is used when required by the robot-objects, it can be configured to sense the environment periodically.

4.3 Non – Periodic Sensors

4.3.1 Overview

Non-periodic sensors are the type of sensors which are used as per requirement. Typically such sensors are used on certain conditions in environments resulting from previous robot actions or information received from other periodic sensors.

In Cooperative Robotic Simulator the Non-periodic sensors are implemented as individual classes that encapsulate the physical characteristics of the sensors and methods. These classes have methods to create and configure requests for sensor information sent to the environment and process the response received from the environment. A request is made for information for specific sensor by calling the appropriate method to send the request. During the execution of this method a sensor request is created with the attributes of the sensor like its type and range and added to the request queue. This request is sent to

the environment which returns the status of the sensor as a response. Bumper sensor and the Heat sensor are examples of non-periodic sensors used in the simulation.

4.3.2 Bumper Sensor

The Bumper sensor is directional and precise sensor that checks for any obstacles next to it. It can be used to detect and maneuver around obstacles. The Bumper sensor is called using the method get_bp(). In this method the relative position of the sensor with the robot is used to create a sensor request that will be sent to the environment. The response received for the request will contain information if the bumper sensor is activated indicating the robot hitting an obstacle. This information is used to modify the state array in the robot-object at the index STATE_BUMPER.

4.3.3 Heat Sensor

The heat sensor is a non-directional sensor that can be used to find the density of objects with heat in an area. A heat sensor request is made in the control code by calling heatSensorCheck() method. In this method a robot request is created containing the range of the heat sensor. The environment calculates the heat that will be sensed by the heat sensor from objects with heat present in the given range. This information is put in the response sent by the environment. It is used to update the state array at index STATE_HEAT.

Chapter 5 Synchronization

5.1 Introduction

The Cooperative Robotic Simulator is a collection of components executing concurrently. These components are brought together into a single application by the Environment using timestamps. As the robot-objects might be running on a different machine at a different rate, synchronization between the Robot-Objects and the Environment is very crucial to the veracity of the simulation. There are many options in maintaining the synchronization based on Timestamp or Event based synchronization models. These can be implemented using busy looping, timers or interrupts.

5.2 Synchronization using Timestep

The synchronization between the various components in the simulator is achieved using timesteps to delineate execution cycles. The simulation is run at a predefined rate. At the beginning of each execution cycle the Environment gathers the commands from all the robots for the current timestep by sending out a timestamp representing the current point of time in the simulation. It then executes these commands and sends back the responses for any sensor requests made. The operations for the current timestep end with the Environment reflecting the changes in the simulation by updating the viewer.

5.3 Abstraction by the Hardware Simulator

As the Robot-Objects are running at a different rate from the Environment, the commands issued by a robot may not be small enough to be executed in one step of the Environment's execution. The hardware simulator plays an important role in maintaining synchronization in this situation. The commands are broken down into pieces before being placed in the request queue appropriate for the current Environment execution cycle. The Hardware Simulator sends the requests for each timestep and the robot resumes after all requests in the last command are executed. The hardware simulator modifies the state array with any responses received for sensor request made, thus abstracting the robots from the actual execution of their commands by the Environment.

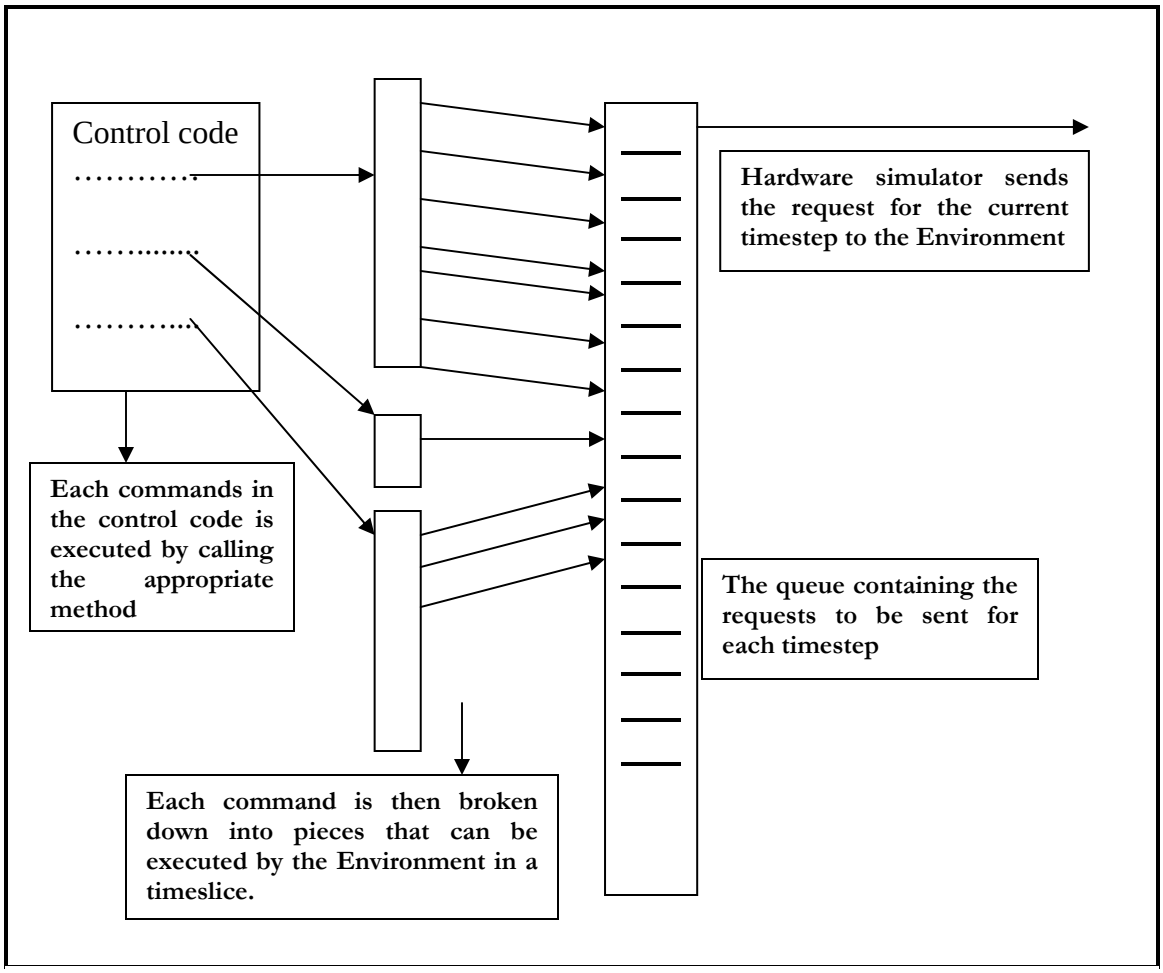

Figure 10: Abstraction by the Hardware Simulator

Chapter 6 Conclusions and Future Work

6.1 Conclusions

This research project was aimed at exploring the scope of a Robotics Simulator to aid in the research of Cooperative Multi-agent algorithms in a complex workspace comprising of many classes of robots and varied environments. The various components in the simulator have been tested to work with building and running simulations of Scout robots in different virtual environments.

All the commands implemented in the Scout class have been utilized successfully in the simulation. This has been verified by visual confirmation of their execution using the viewer during simulation.

Cooperative Robotics Simulator succeeds in laying a framework that satisfies the design requirements such as flexibility and dynamic architecture. The robot class of Scout has been successfully emulated. The simulator is code compatible with the Scout. The fields in the robot-object have the same name and structure as those in the real scout. Hence the commands in the code running on a real scout can be copied without the C headers into the control code class in the simulator. The commands will be executed by the corresponding implementation in the robot-object. Evidently this compatibility is restricted to only the common subset of the languages wherein the source is in C language and the

implementation is in Java. The use of pointers or other C specific data-structures like structures in the control code is therefore not possible.

The simulator has its share of success and has shown areas that require more exploration.

6.2 Future Work

A base has been established using the current implementation on which the following future work can be envisioned.

- A robot definition file can be created that will contain the various physical attributes of the robot and those of its peripherals. In the current implementation the environment is created from information about objects and robots by reading the environment model file. Thus all robot-objects have to connect to the environment in this static architecture for the simulation to start. The robot definition can be shifted without any changes from the environment model file into their own respective files. This makes it easier for robots to add themselves to the environment at anytime during the simulation and still be able to convey all the required information. The protocol between the environment and the robot simulator needs to be modified to adjust to objects dynamically adding themselves to the environment.

- More classes of robots like Pioneers, AmigoBots can be emulated by encapsulating their attributes and operations. The simulator is capable of simulating scout robots.

Other robots can be similarly simulated by implementing supported commands and other methods required to store and modify the State of the robot.

- Other peripherals like robot-arms and grippers, camera, gps can provide more information about the surroundings and capability to handle complex operations. Although existing emulations of peripherals are sufficient to perform operation involving detection and avoidance operations, more sensory information and physical capabilities can be useful to perform a combination of tasks. The peripherals can be emulated by implementing the methods that will create a request for the sensor information or a specific action and the methods that either reflect the requested actions in the simulation or generate the responses in the environment for the sensor requests.

- Adding functions to existing peripherals: Dual frequency sonar Long range LF mode combined with high resolution HF mode and interfaces for log and compass. Existing peripherals can be modified to generate more information and accept more options as input to generate such information. For example the sonar sensor can be used with a fixed range. This can be modified by providing an option to choose from the LF and the HF modes which sets the range of the sensor. Providing a way to set probabilistic degradation in sensor information can also be an important step towards more realistic simulation of a workspace.

REFERENCES

[1] Anonymous, "Nomad Scout User's Manual", Nomadic Technologies Inc., July 12, 1999

[2] Anonymous, "Nomad Scout language Reference Manual", Nomadic Technologies Inc., June12, 1999.

[3] Kumar V. and Sahin F., "Foraging in Ant Colonies applied to the Mine Detection Problem", in the Proceedings of SMCia/03 IEEE International Workshop on Soft Computing in Industrial Applications, pp. 61 – 66, Binghamton, NY, June 23-25, 2003.

[4] Kumar V. and Sahin F., "A Swarm Intelligence Approach for the Mine Detection Problem", Proceedings of the SMC 2002, IEEE International Conference on Systems, Man, and Cybernetics, vol. 3, Tunisia, October 2002.

[5] Scout support website, *http://nomadic.sourceforge.net/production/scout/*

[6] The official project website, *http://www.cis.ksu.edu/~sdeloach/ai/projects/crsim.htm*

Appendix A - USER MANUAL

Writing control code

- **The classes to be modified**

  The control code for the robot is run by the robot simulator by calling the method runcode() in RobotControlCode class. Hence the control code in entirety should be written in this method.

- **List of commands supported**

  The robot simulator currently implements a subset of all the commands of the scout. Although this subset is sufficient for accomplishing complex tasks, it puts a restriction on the different commands that can be copied from existing control code. Here is a list of commands implemented

  - **connect_robot (int id, int model, java.lang.String device, int v)**

    Connects the robot identified by id to the environment.

    In the current simulator implementation

    The robot simulator connects to the environment by using values for address and port of the machine the Environment is currently executing. These values are obtained as command line arguments. Hence the values provided to this method are considered only when the parameters to connect to the environment are NOT provided at command-line.

- **void disconnect_robot (int id)**

  Disconnects the robot identified by id from the environment. In the current implementation disconnection is an invalid request for the Environment.

- **void dp (int x, int y)**

  Sets the current location of the robot as the position (x,y). The values in the state array at indices represented by the constants *STATE_CONF_X* and *STATE_CONF_Y* reflect the current location of the robot.

- **void get_bp ()**

  Method get_bp() obtains a fresh reading for the bumper array of the robot. This method returns after modifying the state array at index represented by the constant *STATE_BUMPER*.

- **void heatSensorCheck ()**

  This method uses the heat sensor described in the definition file to detect other hot objects (robots) in range. The range of the heat sensor can be set in the definition file. The heat sensor receives the amount of heat sensed at current location and updates the value in the state array at STATE_HEAT.

- **void pr (int r_wheel, int l_wheel, int dummy)**

  Method pr moves the robot relative to current position. The current implementation considers only the first parameter (r-wheel) as the distance to move. The value provided is a multiple of .1 inches that the robot has to move forward (linearly) from its current location.

- **void scout_pr (int trans, int steer)**

  The relative translation and rotation from the current robot position Translations are given in 0.1 inches, steers in 0.1 degrees.

- **void sonarSensorCheck ()**

  Method sonarSensorCheck() uses the sonar sensor to detect other objects in range. The range of the sonar can be set in the definition file. The sonar can also be located relative to the robot. The attributes for the peripherals are specified in the environment model file generated by the Environment Model Building tool. A typical definition for the sonar sensor is:

  <sensor>

  <id>2</id>

  <type>sonar</type>

  <position>

  <x-relative>3.2</x-relative>

  <y-relative>0</y-relative>

  <z-relative>0</z-relative>

  <range>6</range>

  <radius>0.1</radius>

  <dir-relative>4.712388980</dir-relative>

  </position>

  </sensor>

  This code excerpt sets the location of the sonar at right hand side of the robot pointing to the left (4. 712388980 = 270^{o}) with a range of 6 units.

- **void tagTarget ()**

  Method tagTarget() will diffuse the heat from any object present in its range possessing heat. Thus given the range of the sensor as 3, all the object in the radius of three from the robot will no more possess any heat.

- **void ws(int x, int y, int z, int dummy)**

  Methos ws() makes the robot go into a wait state until the robot comes to a stand still. This command is used to separate commands so that they can be fully executed, i.e. if a move commands and turn commands have method ws() in between them then the robot will not turn until it had completed the move.

Compiling the application

- **Compiling the robot package**

  If Eclipse IDE is being used then there is no need to compile explicitly as the IDE will compile whenever any modification are made. As the class files created are automatically stored in a separate directory (bin), this method is preferred as this would avoid the confusion from mixing the source files and the class files in one directory.

Running the Application

- **On a stand-alone system**

The application can be run on a single system in the following way.

1) Create an environment definition file and store it in the folder environment. Otherwise an existing file can be modified according to need. This folder exists in the folder hierarchy as

   Robosim → TestLoadFiles → environment

2) Start the environment

   The environment can be started up by the following command

   >java edu.ksu.cis.cooprobot.simulator.environment.Environment

   ../TestLoadFiles/environment/complex-3r.xml

   At any time during the simulation a 2-D viewer (A lightweight viewer can be started up to observe the simulation instead of starting the 3-d viewer).The 2-D viewer can be started up by the following command

   >java edu.ksu.cis.cooprobot.simulator.viewer.Viewer2D localhost 3000

3) Start the robots, the number of robots and their ids are determined by the data in the definition file. When the environment is started-up using the complex-3r.xml definition file as shown above, the environment contains three robots with ids robot0, robot1, robot2. So the robots are started up separately with the following commands:

   >java edu.ksu.cis.cooprobot.simulator.robot.RobotControlPanel robot0 localhost 8000

   >java edu.ksu.cis.cooprobot.simulator.robot.RobotControlPanel robot1 localhost 8000

>java edu.ksu.cis.cooprobot.simulator.robot.RobotControlPanel robot2 localhost 8000

- **In a distributed mode**

The application can be run on in a distributed environment in the following way.

1) Create an environment definition file and store it in the folder environment.

   Otherwise an existing file can be modified according to need. This folder exists in the folder hierarchy as

   Robosim → TestLoadFiles → environment

2) Start the environment

   The environment can be started up by the following command

   >java edu.ksu.cis.cooprobot.simulator.environment.Environment

   ../TestLoadFiles/environment/complex-3r.xml

   At any time during the simulation a 2-D viewer (A lightweight viewer can be started up to observe the simulation instead of starting the 3-d viewer).The 2-D viewer can be started up by the following command

   If a viewer is started on the same machine as the environment, then use the command.

   >java edu.ksu.cis.cooprobot.simulator.viewer.Viewer2D localhost 3000

   Otherwise, if the environment if running on procyon, then the viewer can be started on a different machine by typing the command

   >java edu.ksu.cis.cooprobot.simulator.viewer.Viewer2D procyon.cis.ksu.edu 3000

3) Start the robots, the number of robots and their ids are determined by the data in the definition file. When the environment is started-up using the complex-3r.xml definition file as shown above, the environment contains three robots with ids robot0, robot1 and robot2. When the environment is running on a machine procyon, the robots can be started up separately on different machines with the following commands:

>java edu.ksu.cis.cooprobot.simulator.robot.RobotControlPanel robot0 procyon.cis.ksu.edu 8000

>java edu.ksu.cis.cooprobot.simulator.robot.RobotControlPanel robot1 procyon.cis.ksu.edu 8000

>java edu.ksu.cis.cooprobot.simulator.robot.RobotControlPanel robot2 procyon.cis.ksu.edu 8000