# ROBOSIM -
# PIONEER ROBOT INTERFACE

by

VIKRAM RAMAN

B.E., University of Madras, India, 2003

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

2005

Approved By:

Major Professor
Scott A. Deloach, Ph.D.

## ABSTRACT

This report describes the design and implementation of a robot interface component of the RoboSim. RoboSim is used to perform simulations of many heterogeneous types of robots all working within a single, virtual environment. It provides a distributed framework to develop various robotic applications with different robot species.

This report specifically documents the design and implementation of the application programming interface for the *Activ*Media Pioneer robot, which serves to emulate the behavior of the Pioneer robots in the simulation environment. It also includes interfaces to emulate the various peripherals such as the sonar, heat and laser sensors.

All the features were tested through a simple application that involved controlling the simulated robot within the virtual environment. An analysis of the results and the conclusions drawn are also presented.

# TABLE OF CONTENTS

# LIST OF FIGURES

# ACKNOWLEDGEMENTS

I would like to thank Dr Scott Deloach, Dr David Gustafson and Dr William Hankley for their guidance during the entire course of the project. I would also like to thank my project members Balakumar Krishnamurthi, Thomas Kavukat and Ryan Shelton for all their help and commitment in making this project a success.

# Chapter 1 Introduction

## 1.1 Simulation

Simulators provide users valuable feedback while designing real world systems. They allow the designer of a system to determine the correctness and efficiency of a design before its actual implementation. Consequently the users may explore the merits of alternative designs without actually physically constructing the systems. Also by investigating the effects of specific design decisions during the design phase rather than the construction phase, the overall cost of building the system is reduced significantly.

## 1.2 Cooperative Robotic Simulator

This project involved the design and implementation of the *Activ*Media Robotic Interface for Applications (ARIA) for the Pioneer robot within the framework of our cooperative robotic simulator named RoboSim.

RoboSim is a distributed simulation environment for testing cooperative robotic applications. Robotic applications with arbitrary number of robots working in a complex physical environment can be tested and developed using RoboSim. The simulator consists of different components: the Environment simulator, the `GeometryClient`, the `HardwareSimulator`, the Robot-Object, the Viewer component, the `RemoteControl` component, the `CommunicationsSystem` and the `ControlPanelClient`. The component model of the simulator is shown in Figure 1.

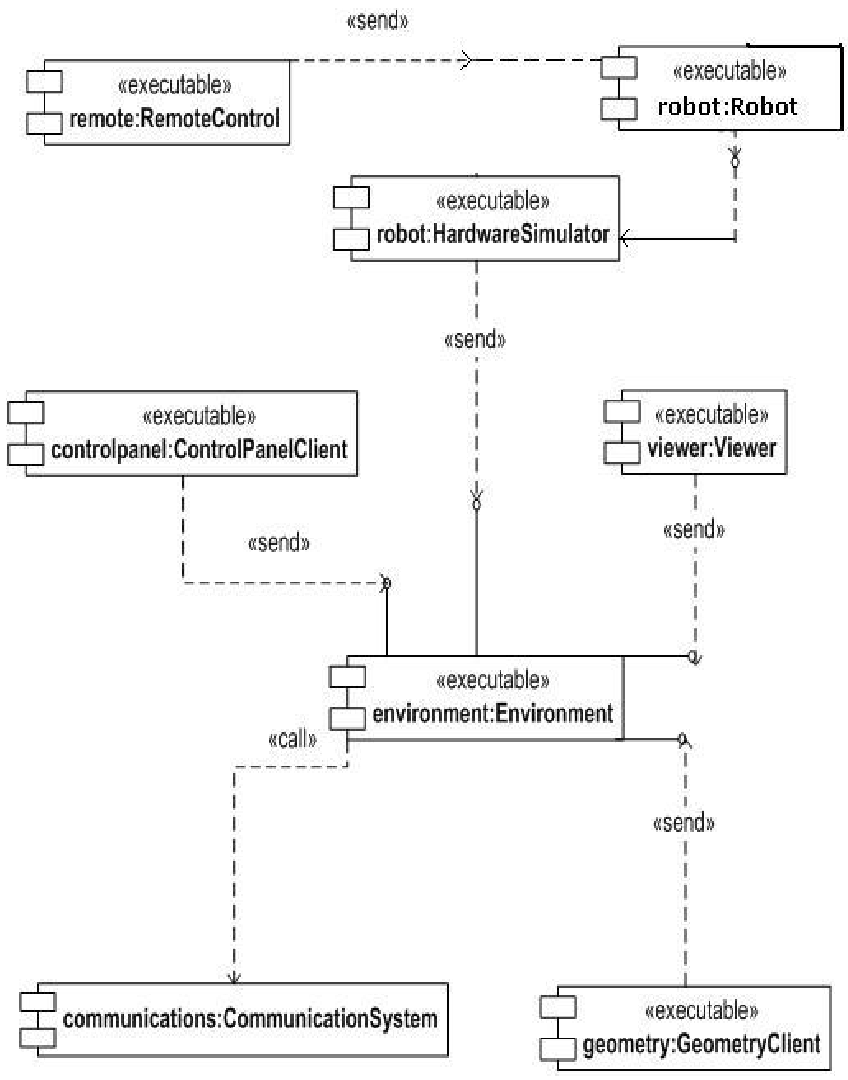


**Figure 1: Cooperative Robotics Simulator Architecture**

### 1.2.1 Environment Simulator

The Environment Simulator is the central component of the simulator. It is responsible for maintaining the state of all the components of the system including the robots. All the other modules interact through the environment. It receives the sensor and motion requests from the robot simulator each time step and provides the simulated robot with the sensor response. The environment also updates the viewer module with the current state of all the objects within the simulation environment each time step.

### 1.2.2 `GeometryClient`

The `GeometryClient` is responsible for the geometric representation of the environmental objects, collision detection and distance finding. By finding intersection between objects, it can prevent objects from overlapping in the environment and also simulate sonar and laser range-finding. It simulates the virtual environment within which the entire simulation is performed. The environment translates requests from the `HardwareSimulator` into appropriate messages to the `GeometryClient`.

### 1.2.3 Robot Simulator

The Robot Simulator comprises of three parts: the HardwareSimulator, the Robot-Object and a robot control code, which will contain the robotic application logic. A standard application programming interface (API) is defined between the robot control program and the `HardwareSimulator`. This API facilitates the robot control code to work with various Hardware Simulators. In reality the sensors and actuators exist on the robot hardware, while in the Robot simulator the `HardwareSimulator` emulates the standard sensors and actuators.

#### 1.2.3.1 `HardwareSimulator`

The `HardwareSimulator` is as an interface between the Robot-Objects and the Environment. It is a layer of abstraction emulating the hardware of the simulated robot such as the sonar, heat, laser sensors and also the robot itself. It processes and sends the robot sensor, motion requests to the environment every time step, and passes the environment responses back to the Robot-Object.

#### 1.2.3.2 Robot-Object

The Robot-Object encapsulates the API to the simulated robot. An API for the Pioneer robots was implemented as a part of this project. There also exists an API for the Scout class of robots [4]. The Robot-Object provides the ability for an application programmer to control the robot behavior, the various sensors of the robots and access the data returned by the robot sensors. The robot interface exists above the `HardwareSimulator` layer and thus interacts with the environment module through the `HardwareSimulator`.

### 1.2.4 Viewer component

The RoboSim viewer displays a two dimensional or a three dimensional view of the simulation to the user. This is an integral component of the simulator as it allows the user to observe the behavior of the environment objects and the robots within the simulation environment. The viewer includes the capability to customize the view in terms of the camera angle, the zoom level and the light source. The simulation can also be recorded and replayed using the viewer. The environment updates the viewer with the current state of all the environment objects at every time step of the simulation.

### 1.2.5 `RemoteControl`

The `RemoteControl` component provides the capability to control a simulated robot from a remote machine. It includes a graphical user interface (GUI) for the end user to view an individual robot's parameters. It contains graphical buttons that provide the ability to control the simulated robot manually. The action events associated with each of the buttons generate the pertinent motion or sensor requests to the Robot-Object. It also consists of an embedded viewer that can be used to view the entire simulation. Thus the `RemoteControl` component presents all the aspects of the simulation to the end user through a single, convenient GUI.

### 1.2.6 `ControlPanelClient`

The `ControlPanelClient` is a graphical user interface that allows the user to manipulate the environment. It is possible to load the environment files, start and stop a simulation and set the various simulation properties; these include the network options, the communication settings and the factors in the simulation. It allows the user to monitor, change the status of the simulated robots within the environment and includes 2D, 3D representations of the simulation environment.

### 1.2.7 `CommunicationsSystem`

Cooperative robotic applications involve a large amount of communication between the various robots. The `CommunicationsSystem` allows control over the delivery of messages by providing various standard communication capabilities such as broadcasting, multicasting and point to point messaging.

Chapter 2 will review the Pioneer robot and its associated client software in detail; we will then look at the different components of the Robot package in Chapter 3 followed by the conclusions and possible future enhancements in Chapter 4.

# Chapter 2 Pioneer

## 2.1 Introduction

This project was aimed at simulating the Pioneer class of robots in RoboSim. The *Activ*Media Pioneer is a family of mobile robots, both two wheels and four wheel drive that employs common client-server robotics control architecture [2]. *Activ*Media Pioneer robots can be configured with sonar and laser range finders that provide object detection and range information for collision avoidance, localization, and navigation. The Pioneer robot can be controlled remotely using client software or operated manually using a joystick. The main robot-control client software available for the Pioneer robots is the *Activ*Media Robotics Interface for Applications (ARIA), SRIsim *Activ*Media Robot simulator and SRI's Saphira client development suite. The *Activ*Media Pioneer robots operate as the server in a client-server environment. Specifically this project involved the implementation of the ARIA API as applicable to RoboSim to enable the testing of applications that would utilize the Pioneer class of robots.

## 2.2 ARIA

ARIA is an object-oriented, robot control applications-programming interface for *Activ*Media Pioneer robots. It is a C++-based development environment that provides a high performance access to and management of the robot server, as well as to the many accessory robot sensors and effectors. There also exists a Java based API that employ native method calls for the robot control applications to use. It includes clear and convenient interfaces for applications to access and control *Activ*Media Robotics accessory sensors and devices, including operation and state reflection for sonar and laser range finders, inertial navigation devices, and many others. It is an ideal platform for robotics client applications development.

The ARIA API consists of a set of classes that collectively provide an interface for the client side application to control the robot and its peripherals. These include access to the current state of the robot, its configuration and the readings of the sensors that can be used for obstacle avoidance and to control the movement of the robot. The most important class is ARIA's `ArRobot` class which collects and organizes the robot's operating states, and provides a clear and convenient interface for other ARIA components, as well as upper-level applications, to access the robot state reflection information for purposeful control of the robot and its accessories. The `ArRangeDevice` class provides a complete set of interfaces to access the histories of relevant readings associated with each sensor. The `ArSonarDevice` and the `ArSick` are two of its subclasses that represent the sonar and the laser range finder respectively. All range devices are range-finding devices that periodically collect 2-D data at specific global coordinates. The `ArSensorReading` class holds the data, with each sensor (range device) having an instance of this class to represent its associated readings. A high level class diagram representing the interaction of these classes is given in Figure 6. The commands supported by the different ARIA classes are presented below.

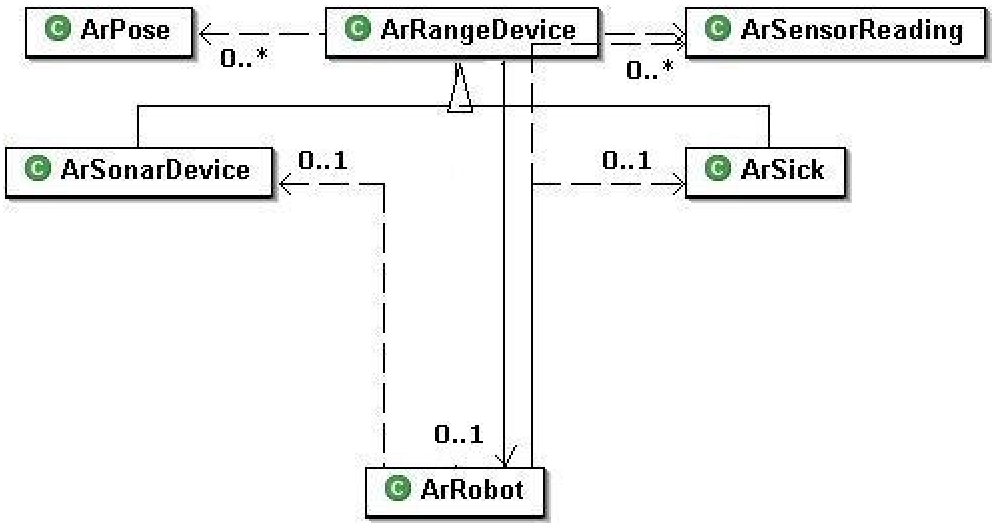


**Figure 2: High Level class diagram of ARIA**

## 2.3 Class `ArRobot`

The `ArRobot` class acts as the client-server communications gateway, central database for collection and distribution of robot information, and systems synchronization manager. This class is the gathering point for range-finding sensor and Actions classes. It maintains and distributes a snapshot of the robot's operating conditions and values. Low-level sonar readings are also reflected in `ArRobot` and can be examined with the relevant methods provided. It handles the low-level details of constructing and sending the client-command packets to the robot as well as receiving and decoding the various Server Information Packets from the robot. The commands supported by the `ArRobot` class for robot control is given below.

### 2.3.1 Client-Server connection commands

`bool blockingConnect()`

Connects to the robot, returning only when a connection has been established or when a connection can't be made. This function returns true if the connection is successfully established.

`bool disconnect()`

Disconnects the client from the robot. Returns true if the disconnection was successful.

`bool isConnected()`

Questions whether the client is connected to the robot or not, returning true if connected to the robot.

### 2.3.2 Motion commands

`void move(double distance)`

Moves the robot forward/backward by the given distance, where the distance is specified in mm. The robot moves at a translational velocity that has been set by the most recent call to the setVel() command.

`void stop()`

Stops the robot, resets the translational and rotational velocity to zero

`void setRotVel(double velocity)`

Sets the rotational velocity of the robot in degrees/sec. Future calls to the setHeading() command, will turn the robot to the desired heading at a rotational velocity that has been set here.

**`void setVel(double velocity)`**

Sets the translational velocity of the robot in mm/sec. Subsequent move instructions will move the robot at this velocity.

**`void setVel2(double lvelocity, double rvelocity)`**

Sets the velocity in mm/sec of each of the wheels on the robot independently

**`void setMaxRotVel(double velocity)`**

This sets the maximum rotational velocity the robot will go in degrees/sec (must be a non-zero number)

**`void setMaxTransVel(double velocity)`**

This sets the maximum translational velocity the robot will go in mm/sec.

**`void setHeading(double heading)`**

Sets the desired absolute heading of the robot in degrees. The ArRobot class does not contain any explicit turn instructions; this instruction turns the robot to the desired heading. The robot turns at a rotational velocity specified by the most recent call to the setRotVel() instruction.

**`void setDeltaHeading(double deltaHeading)`**

Sets the heading relative to the current heading. For example, if the current heading was 180 degrees, the command `setDeltaHeading(-90)` would set the heading of the robot to 90 degrees.

### 2.3.3 Range devices related commands

**`void addRangeDevice(ArRangeDevice device)`**

Adds a range device to the robot's list of range devices, and sets the device's robot pointer

**`void remRangeDevice(ArRangeDevice device)`**

Removes a range device from the robot's list of range devices

**`int getClosestSonarRange(double startAngle, double endAngle)`**

Returns the closest of the current sonar readings in the given range (in mm)

**`int getClosestSonarNumber(double startAngle, double endAngle)`**

Returns the number of the sonar that has the closest current reading in the given range

**`ArSensorReading getSonarRange(int sonarNumber)`**

Gets the range of the last sonar reading for the given sonar.

**`ArSensorReading getSonarReading(int sonarNumber)`**

Gets the sonar reading for the given sonar

**`bool isSonarNew(int sonarNumber)`**

Find out if the given sonar has a new reading. Returns false if the sonar reading is old, or if there is no reading from that sonar.

## 2.4 Class `ArRangeDevice`

Range devices are abstractions of sensors for which there are histories of relevant readings. A range device is attached to the robot with `ArRobot.addRangeDevice` and removed with `ArRobot.remRangeDevice`. `ArSonarDevice` and `ArSick` are two of the subclasses of `ArRangeDevice` representing the sonar and the laser respectively. In ARIA the sonar readings automatically come included with the standard server information packets and is processed by the `ArRobot` class. The sonar must be explicitly added with the Robot-Object to use the sonar readings for control tasks. This class has two buffers, a current buffer for storing just recent (relevant) readings, and a cumulative buffer for a longer history. The sizes of both can be set in the constructor. The set of relevant methods in the `ArRangeDevice` class is given below.

### 2.4.1 Robot related commands

**`void setRobot(ArRobot robot)`**

Sets the robot this device is attached to

**`ArRobot getRobot()`**

Gets the robot this device is attached to

### 2.4.2 Buffer related commands

**`void addReading(double x, double y)`**

Adds a reading to the buffers

**`void setCurrentBufferSize(int size)`**

Sets the size of the current readings buffer

**`void setCumulativeBufferSize(int size)`**

Sets the size of the cumulative readings buffer

**`void setMaxRange(double range)`**

Sets the maximum range for this device

**`double getMaxRange()`**

Gets the maximum range for this device

**`std::list<ArPose *> *getCurrentBuffer()`**

Gets current buffer of readings.

**`std::list<ArPose *> *getCumulativeBuffer()`**

Gets the cumulative buffer of readings

## 2.5 Class `ArPose`

This class represents a position in the environment. It is closely related to the `ArRangeDevice` class, as every element in the current and cumulative buffer is an instance of this class. The commands supported by this class is given below.

**`double getX()`**

Gets the X value of the position

**`double getY()`**

Gets the Y value of the position

**`double getTh()`**

Gets the heading of the position

**`double findDistanceTo(ArPose position)`**

Finds the distance of the given position to the current position.

## 2.6 Class `ArSensorReading`

A class to hold a sensor reading; should be one instance per sensor.

### 2.6.1 Sensor reading commands

**`bool isNew(int counter)`**

Given a counter, returns whether the reading is new

**`int getRange()`**

Gets the range of the sensor reading

**`double getX()`**

Get the x position of the reading

**`double getY()`**

Get the y position of the reading

**`void newData(int range, int x, int y, int counter)`**

Updates the reading with new data.

# Chapter 3 Robot Package

## 3.1 Introduction

The Robot package is a part of the hierarchical framework in RoboSim that contains the classes that implement the functionality of the Robot Simulator that was briefly described in Chapter 1. The Robot Simulator is responsible for simulating the different classes of robots such as the Scout and the Pioneer in RoboSim. This chapter will present the overall architecture of the robot package and the Pioneer Robot-Object in detail.

## 3.2 Architecture

The generic robot package is at the top of the hierarchy. It contains the `AbstractRobot` class that defines operations that are common to all the classes of robots. A `PeriodicSensor` class handles the generation of events at the frequency the periodic sensor being emulated is to be used. The `RobotRequest` and `RobotSensorResponse` classes encapsulate the information present in the requests being sent to the Environment and in the corresponding sensor responses that is returned. 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.

Next in the hierarchy are the packages emulating the Scout and the Pioneer robots. These robot specific packages contain the Robot-Objects that implement the abstract methods common to all the Robot-Objects as specified in the `AbstractRobot` Class along with the respective commands supported by the particular type of robot that they emulate.

Each Robot-Object is associated with an instance of the `HardwareSimulator` in order to emulate its sensors and other peripherals. The Robot-Object component facilitates intelligent multi-agent communication, navigation and localization. The Robot-Object encapsulates the features and actions of a specific type of robot in the simulation environment. The generic implementation of the Robot-Objects allows the simulation of varied robots with different sets of sensors and actuators. We will now look at some of the classes and relevant methods from the ARIA package that was implemented in RoboSim.

## 3.3 Class `ArRobot`

The `ArRobot` class encapsulates the Pioneer class of robots discussed in Chapter 2. It maintains and distributes a snapshot of the simulated Pioneer robot's operating conditions and values to the client. It provides an application programming interface to the simulated Pioneer robot within the framework of RoboSim. The association between the different classes of the robot package and the `ArRobot` class is given in Figure 3 and the class diagram of the `ArRobot` class is given in Figure 4.

The `ArRobot` class supports a set of commands to initially connect to the simulated robot, to configure its sensors, observe the corresponding sensor readings and to control the motion of the robot according to the application logic. The main set of commands that were implemented in RoboSim is given below.

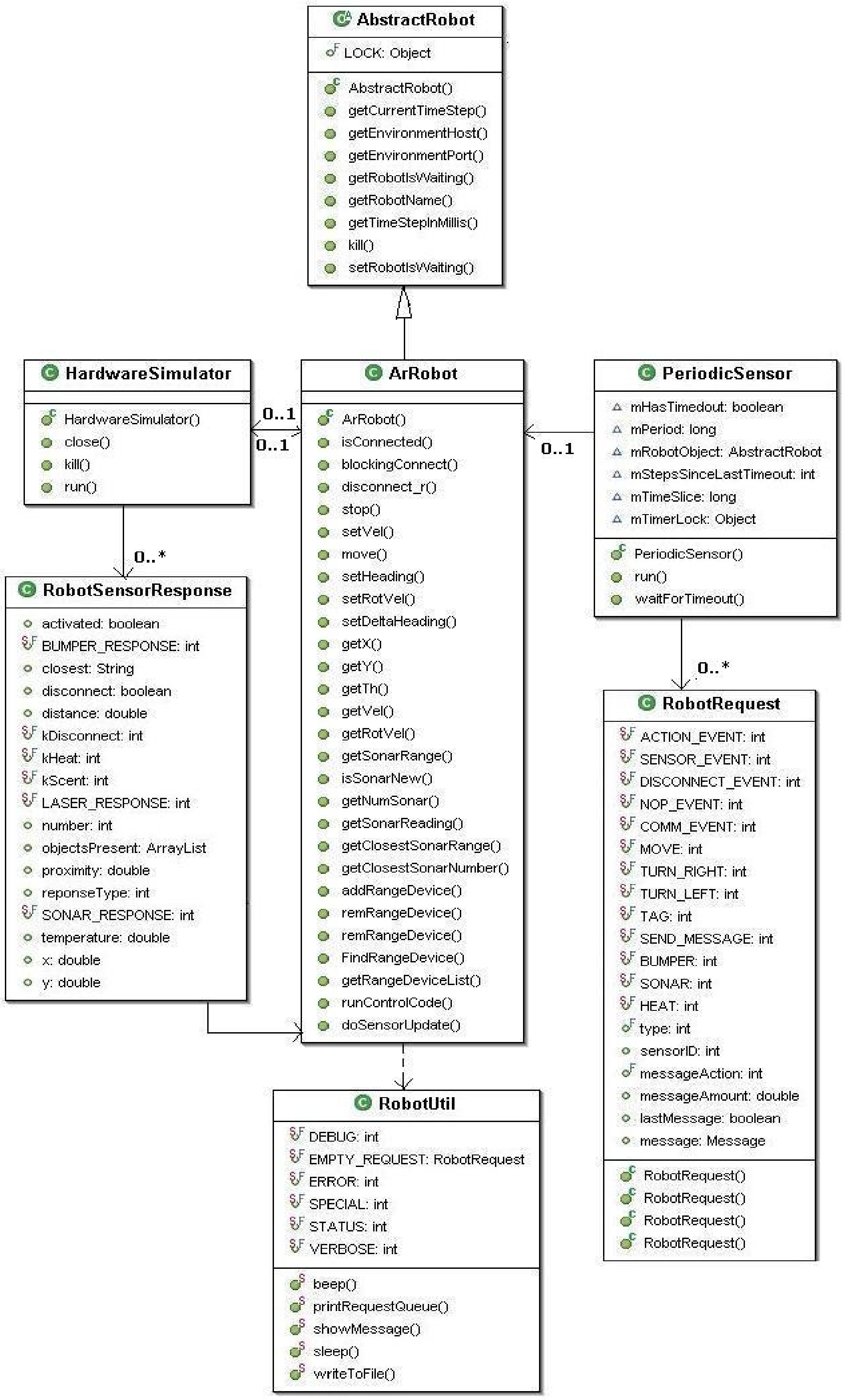


**Figure 3: Class diagram of the robot package**

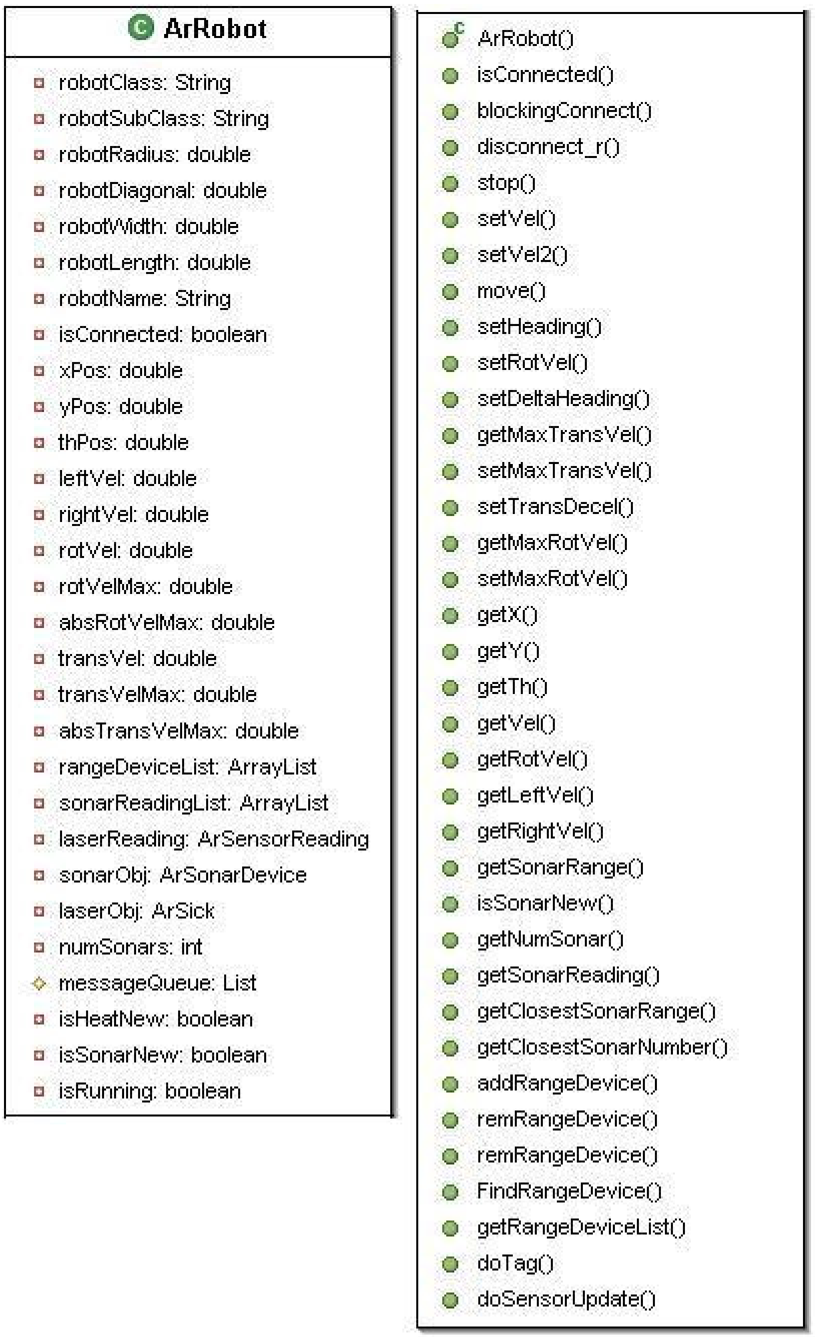


**Figure 4: ArRobot class diagram**

- Client-Server Connection Commands
    - `boolean blocking_connect()`
    - `boolean disconnect()`
- Motion Control Commands
    - `void move(double distance)`
    - `void stop`
- Robot Parameter Setting Commands
    - `void setRotVel(double velocity)`
    - `void setVel(double velocity)`
    - `void setHeading(double heading)`
- Range Devices related commands
    - `void addRangeDevice(ArRangeDevice device)`
    - `void remRangeDevice(ArRangeDevice device)`
    - `int getClosestSonarRange(double startAngle, double endAngle)`
    - `int getClosestSonarNumber(double startAngle, double endAngle)`
    - `double getSonarRange(int sonarNumber)`
    - `ArSensorReading getSonarReading(int sonarNumber)`
    - `boolean isSonarNew(int sonarNumber)`

The `ArRobot` is the most important class in ARIA. It interacts with the other classes in the ARIA package to provide a single convenient interface for the client to observe the current state of the robot and its associated sensors. The class diagram for the entire set of classes in the ARIA package that were implemented in RoboSim is given in Figure 5.

## 3.4 Class `ArRangeDevice`

This class describes a set of methods that are common to all range devices such as the sonar and the laser. Each range device has a history of relevant readings associated with it; the range device class provides two buffers to store these readings – the current buffer and a cumulative buffer. As the name suggests the current buffer consists of most recent readings of the respective range device. The cumulative buffer contains the readings of the range device for a period of time as defined by the size of the cumulative buffer. `ArSonarDevice` and `ArSick` are two of the subclasses of `ArRangeDevice` specifically representing the sonar and the laser respectively. Some of the commands supported by the `ArRangeDevice` class that were implemented are given below.

- Robot related commands
    - `void setRobot(ArRobot robot)`
    - `ArRobot getRobot()`

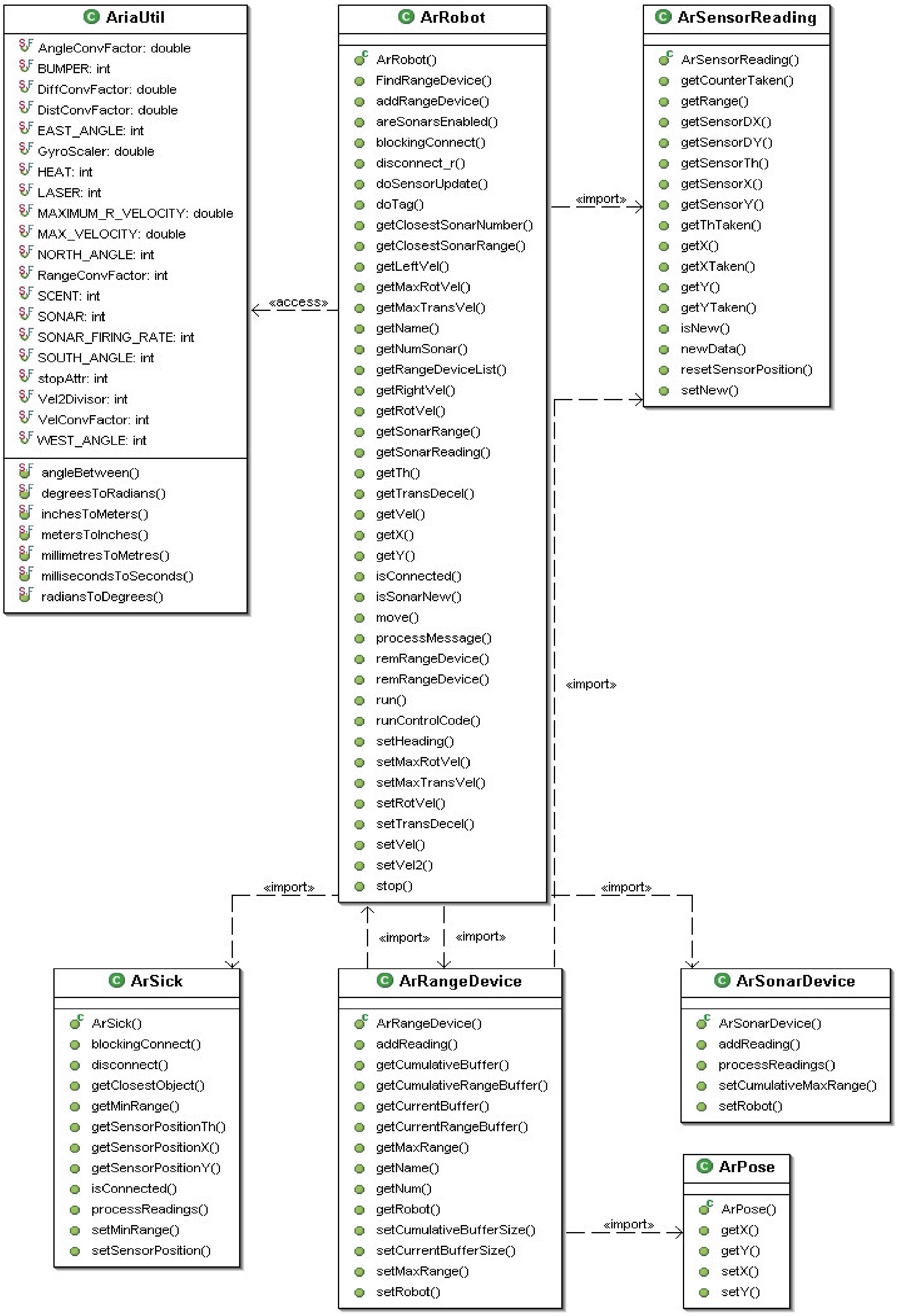


**Figure 5: Aria class diagram**

- Device buffer commands
    - `void addReading(double x, double y)`
    - `void setCurrentBufferSize(int size)`
    - `void setCumulativeBufferSize(int size)`
    - `ArrayList getCurrentBuffer()`
    - `ArrayList getCumulativeBuffer()`

## 3.5 Class `ArPose`

This class represents a position in the environment. Every range device reading has a x and y position associated with it along with the heading. All positions in the ARIA API are represented in terms of this class. Thus all the buffer readings in the `ArRangeDevice` class are represented as an instance of the `ArPose` class. The set of methods implemented as a part of this class is given below.

- Position related commands
    - `double getX()`
    - `double getY()`
    - `double getTh()`
    - `double findDistanceTo(ArPose position)`

## 3.6 Class `ArSensorReading`

This class encapsulates a sensor reading. Each sonar in the sonar array has an instance of the `ArSensorReading` class associated with it. Data such as the range returned by each sonar and the x and y position of the reading can be accessed through this class. The set of relevant methods that were implemented is given below.

- Reading related commands
  - `bool isNew(int counter)`
  - `int getRange()`
  - `double getX()`
  - `double getY()`
  - `void newData(int range, int x, int y, int counter)`

We will now look at the `HardwareSimulator` class in the robot package and how it interacts with the ARIA package that was presented in order to simulate the Pioneer robot in RoboSim.

## 3.7 `HardwareSimulator`

The `HardwareSimulator` class emulates the various sensors and actuators on a robot. It acts as interface between the `ArRobot` class and the Environment module. Each Robot-Object contains an instance of the `HardwareSimulator`; The Robot-Object adds any sensor or motion requests to the request queue, when a time step is received the requests for that time step are removed from the request queue and sent to the environment by the `HardwareSimulator`. The `HardwareSimulator` then waits for a response for each of those requests and notifies the Robot-Object of the sensor response. A sequence diagram depicting the interaction between the `ArRobot` class, the `HardwareSimulator` and the environment is show below in Figure 6.

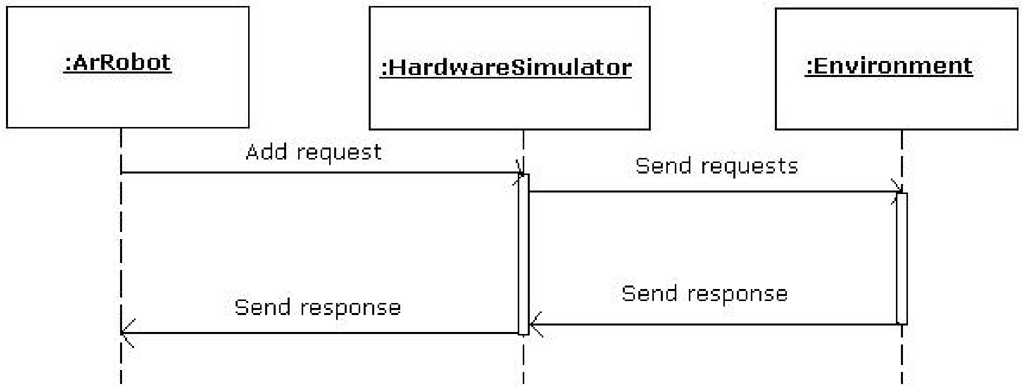


**Figure 6: Sequence diagram for interaction between ArRobot, HardwareSimulator and Environment**

## 3.8 Communication Protocols

The Robot simulator module consists of two main communication protocols that dictate the communication between the Robot-Object, the `HardwareSimulator` and the Environment. The protocol between the Robot-Object and the `HardwareSimulator` controls the transfer of the robot commands onto the `HardwareSimulator`. The protocol between the `HardwareSimulator` and the Environment controls the transfer of the commands from the `HardwareSimulator` onto the Environment, which in turn reflects the effects on the virtual simulation environment.

### 3.8.1 Protocol between Robot-Object and Hardware Simulator

The Robot-Object generates requests that need to be serviced each time step by adding it to the r`obotRequestQueue`. The `robotRequestQueue` is a data structure that is maintained by the `HardwareSimulator` and it contains all the robot requests that need to be processed and sent to the environment at the beginning of each time step. The `HardwareSimulator` abstracts the processing of these requests and sends the response back to the Robot-Object. The Robot-Object waits till the `HardwareSimulator` notifies it of a sensor-response. The sequence diagram for the protocol between the `HardwareSimulator` and the Robot-Object is given in Figure 7.

As can be seen from the figure, the Robot-Object first instantiates a `HardwareSimulator` thread to run on its behalf in order to emulate the robot peripherals. The first thing that the `HardwareSimulator` does upon instantiation is to establish a connection with the environment. It then periodically monitors the robot requests queue, and sends any of the robot's requests to the environment. The `HardwareSimulator` then waits for the environment to process the requests and deliver the sensor responses. These sensor responses are then sent back to the Robot-Object. Thus the Pioneer robot is simulated in RoboSim.

### 3.8.2 Protocol between Hardware Simulator and Environment

The `HardwareSimulator` interfaces the Robot-Object with the environment. Every time step, it receives the current time step of the simulation from the environment, sends the relevant robot requests to the environment and waits for the sensor responses from the environment. A sequence diagram depicting the protocol between the `HardwareSimulator` and the Environment is given in Figure 8.

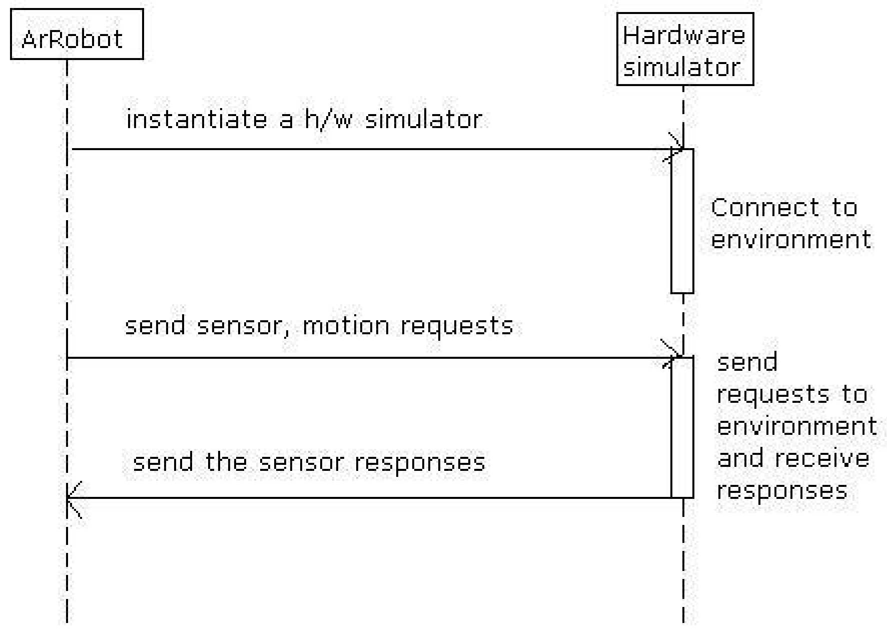
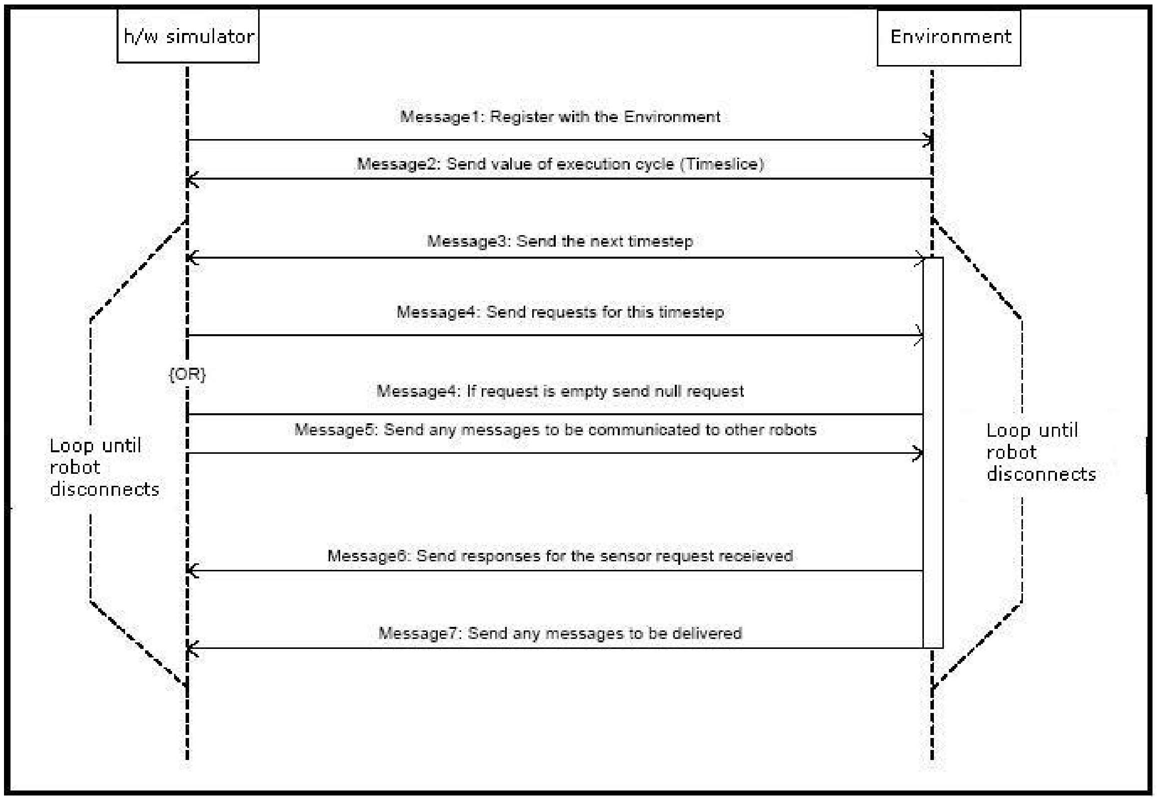

ArRobot
Hardware simulator
instantiate a h/w simulator
Connect to environment
send sensor, motion requests
send requests to environment and receive responses
send the sensor responses

**Figure 7: Sequence diagram for protocol between ArRobot and Hardware Simulator**



h/w simulator
Environment
Message1: Register with the Environment
Message2: Send value of execution cycle (Timeslice)
Message3: Send the next timestep
Message4: Send requests for this timestep
{OR}
Message4: If request is empty send null request
Message5: Send any messages to be communicated to other robots
Loop until robot disconnects
Loop until robot disconnects
Message6: Send responses for the sensor request receieved
Message7: Send any messages to be delivered

**Figure 8: Sequence diagram for protocol between Hardware Simulator and Environment**

# Chapter 4 Test Simulation

## 4.1 Introduction

The implementations of the ARIA classes used to emulate the Pioneer robot in RoboSim were tested using a simple robotic application. The application involved the simulation of many Pioneer robots navigating autonomously through a maze in search of a simulated Gold object in the environment. The robots navigated thorough the maze using only the sensor readings of the simulated robot. We will now look at the structure of the maze application.

## 4.2 Maze application

The basic design of the maze application is given below. The `GoldFinder` class is the main class of the application. Its main function is to start a given number of Robot-Object threads. The `Gold` class extends the `ArRobot` class and thus represents the Robot-Object in this application. Each Robot-Object thread executes the robot control code until either of the robots has found the Gold. The `MazeMessage` class is used to encapsulate the message information that is passed between the robots in the application. The robot control code involves each robot navigating autonomously through the maze using only the sensor readings obtained through the related methods in `ArRobot`. When any one of the robots has identified the Gold it sends a message to the other robots through the environment and the simulation ends. The simulation utilized all the classes that were implemented as a part of the ARIA package and all the implementations were tested successfully.

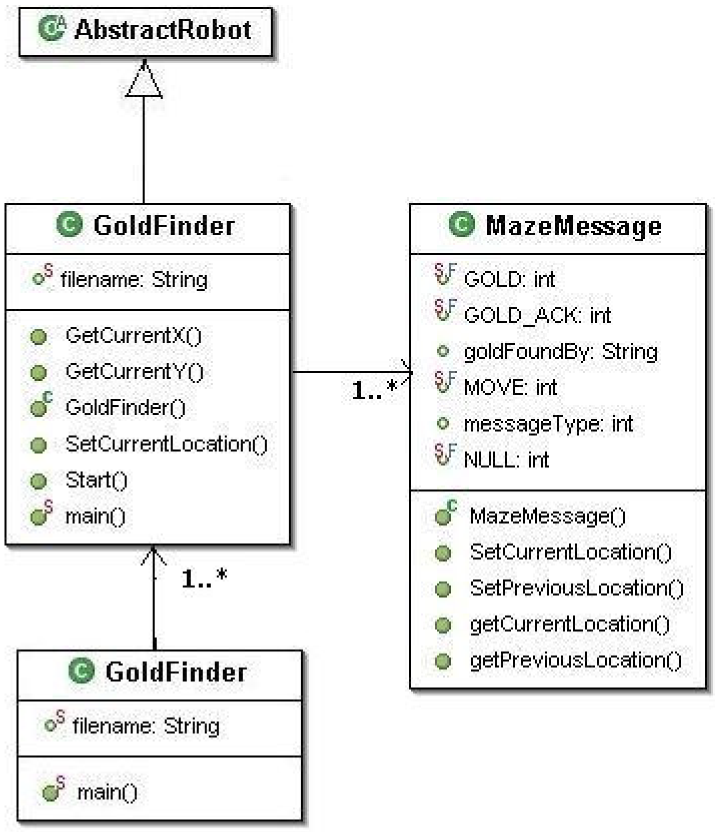


**Figure 9: Maze application**

# Chapter 5 Conclusions and Future Work

## 5.1 Conclusion

This project was aimed at expanding the scope of RoboSim to aid running simulations of the Pioneer class of robots in different virtual environments. The ARIA client interface to the Pioneer robot was implemented to this effect. A minimal subset of the ARIA software, related to the motion control of the robot and those providing access to its sensor readings were implemented and utilized successfully in test simulations. The functionalities to interface with the Pioneer robot were thus emulated successfully.

## 5.2 Future Enhancements

A minimal set of classes of the ARIA software have been implemented providing a basis for testing of applications that use the Pioneer robot. Some possible extensions include:

- The physical positions of the sonar and other robot peripherals are currently hard coded in the environment module. The Pioneer robot interface supports the loading of robot parameter files on the robot side, but currently this information is not passed on to the environment module. Also since the emulation of the Scout robot does not include a parameter file being loaded on the robot side, an obvious extension would involve minor modifications in the Scout robot implementation and in the protocol between the `HardwareSimulator` and the Environment, wherein the relevant robot parameters are passed on to the Environment before the start of the simulation. This would facilitate accurate dynamic positioning of the peripherals within the Environment during the entire period of the simulation.

- The Environment waits for all robots to connect to it at the start of the simulation, and the simulation ends once any of the robots disconnect from the Environment. From the perspective of the robotic application designer, it might prove useful to support dynamic addition and removal of robots at any point of the simulation.

- The Environment current does not currently support the moving of robots in the backward direction. But the Pioneer robot interface does support moving the robot in the backward direction, and this has currently been implemented by turning the robot around by 180 degrees and moving it forward the desired distance and resetting the robots heading to the original direction. But this implementation will not provide accurate readings for the sensors when the robot is moving backward. The Environment could thus be modified to allow the moving of robots in the backward direction so as to accurately represent the features of the actual robot.

- Other classes of the ARIA package such as the Action and behavioral classes, classes that provide an interface to peripherals like the Gripper, the Camera etc. can be implemented along with support for such peripherals on the Environment module to enhance the capability of the simulator.

# REFERENCES

[ 1 ] Anonymous, " ARIA Reference Manual Version 1.1.11", Jan 2003.

[2] Anonymous, "Pioneer 3™ & Pioneer 2™ *H8*-Series Operations Manual, version 3", Aug 2003.

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

[4] Rapaka V., "Co-Operative Robotic Simulator – Robot Simulator", May 2004.

# Appendix – User Manual

The `edu.ksu.cis.cooprobot.pioneer.aria` package contains the classes that were implemented as a part of this project. The relationship between the classes themselves are similar to those that exist in the ARIA software, and hence the ARIA user manual should be a good starting point for an application designer wanting to use the classes in this package to control the Pioneer robot. This user manual specifically contains instructions on how to compile the code and run the applications once they have been implemented.

## 1. Robot parameter XML file

The `ArRobot` constructor includes a filename as one of its arguments. This filename is the robot parameter file represented in an xml format. The file itself could be located anywhere; the entire path including the file name could be passed as the argument.

An example of the robot-param XML file is given below. The nodes in the XML file are fixed and if modified, the method `processInfo()` in Class `ArRobot` should also be modified accordingly. Most of the fields in the XML file are self-descriptive. It can be seen that the position of the sonar on the robots have been commented. As mentioned as a part of the future enhancements, if the protocol between the environment and the `HardwareSimulator` were modified, this information can be sent to the environment module in order to represent the sensor positions relative to the robot more accurately.

```
<?xml version="1.0" encoding="utf-8"?>
<robot>
      <params>
            <class>Pioneer</class>
            <subclass>p3at</subclass>
            <!-- radius in mm -->
            <robotRadius>500</robotRadius>
            <!-- half-height to diagonal of octagon -->
            <robotDiagonal>120</robotDiagonal>
            <!-- width in mm -->
            <robotWidth>505</robotWidth>
            <!-- length in mm -->
            <robotLength>626</robotLength>
            <!-- absolute maximum rotational velocity degrees / sec -->
            <maxRVelocity>300</maxRVelocity>
            <!-- absolute maximum mm / sec -->
            <maxVelocity>1200</maxVelocity>

            <!-- Section Sonar parameters -->
            <!-- number of sonar on the robot -->
            <sonarNum>16</sonarNum>
  <!-- SonarUnit <sonarNumber> <x position, mm> <y position, mm>
<heading of disc, degrees>
            <sonar>
              <sonarUnit>
                <sonarNumber>0</sonarNumber>
                <sonarX>147</sonarX>
                <sonarY>136</sonarY>
                <sonarTh>90</sonarTh>
              </sonarUnit>

              <sonarUnit>
                <sonarNumber>1</sonarNumber>
                <sonarX>193</sonarX>
                <sonarY>119</sonarY>
                <sonarTh>50</sonarTh>
              </sonarUnit>
            .
            .
            .
              <sonarUnit>
                <sonarNumber>15</sonarNumber>
                <sonarX>-144</sonarX>
                <sonarY>136</sonarY>
                <sonarTh>90</sonarTh>
              </sonarUnit>
-->
      </params>
</robot>
```

## 2. Environment XML File

When designing an application, an environment model XML file needs to be provided as a command line input to the environment. This would contain the specifics of the virtual environment for the simulation run. Such files are self-descriptive and can be found under the testLoadFiles->environment directory within the Robosim project directory. More information on these can be found in [4]

## 3. Running the application

The code required in order to run an application are:

- edu.ksu.cis.cooprobot.simulator.environment.Environment.java
- edu.ksu.cis.cooprobot.simulator.robot.pioneer.aria.ArRobot.java
- edu.ksu.cis.cooprobot.simulator.viewer.Viewer2D.java

A robotic application can be run using the simulator as follows,

1) Create an environment model file in the following directory,

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/file.xml*

At any time during the simulation a 2-D viewer can be started,

```
>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 xml definition file as shown above, the environment will wait until the number of robots specified in the file has connected to it. The robot(s) can be started as follows

```
>java edu.ksu.cis.cooprobot.simulator.<robotApplication> robotName localhost portNumber
```

The server, ports can be modified accordingly to run the application components in a distributed manner.