Cooperative Mobile Robots Simulation Engine for the Neko Distributed Systems Prototyping Framework

By Smath Sangsubhan

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Xavier Defago

March, 2009

Cooperative Mobile Robots Simulation Engine for the Neko Distributed Systems Prototyping Framework

By Smath Sangsubhan (710037)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Xavier Defago

and approved by
Associate Professor Xavier Defago
Professor Mineo Kaneko
Research Professor Tomoji Kishi

February, 2009 (Submitted)

Abstract

Programming groups of robots and ensuring their proper interactions and coordination is extremely complex and still poorly understood. The difficulty of developing robots system is due to two major factors: First, there is little control on the environment and real-world experiments are too costly (e.g., simulating earthquake environment for rescue robots). Second, there is little programmatic support for development and reuse of specialized software components and protocols, which make development of robots system become time-consuming and difficult to maintain. Hence, there is a strong need for prototyping tools as support for both research and application development.

Simulator for robots or distributed system is not a new study field. Currently a great number of them are existing. We can separated those simulators in to two groups.First group, Robot simulator, most of them provide simulation environment for robot movement and sensor network system in 2D or 3D graphical animation. However, their main concern is only about robots motions. As a result, Implementation of communication mechanism is very difficult and time-consuming. Second group, Network simulator. These simulator allow us to do the simulation of communication on network and help us research in many things such as, message delay or the bottom neck of network. Even though some of them provide support for mobile ad-hoc network, it is still difficult to fully implement mobile robots application on them. In order to evaluate a simulation of distributed algorithm on mobile robots application, we need a simulator that fully support both of mobility and communication.

This has became an inspiration to my research. It was a starting point of the idea to present a new rapid prototyping tool for evaluating distributed algorithm on cooperative mobile robots system.

Contents

Chapter 1

Introduction

Recently, mobile robots system plays an importance role for supporting human in many ways. For instance, dangerous and high-risk works such as mining or rescue support. Or it might be a routine work that make our daily life more comfortable such as factory work or cleaning. However, development of robots system is never be easy and very time-consuming. Then a question that arises is, How can we effectively develop mobile robots system? This became my inspiration and my ultimately goal to provide an effective rapid prototyping approach.

From year to year, mobile robots system has been intensively researched and being used in many field of works. Nowadays, robots are capable of doing many things even some difficult works that human could not do. Consequently, in order to let robots perform those tasks, the implementation of complex algorithms is unavoidable. Moreover, considering cost/performance, the cooperation between multiple robots may be implemented in some situation. However, more complexity in mechanism means more time-consuming in development. So, in order to evaluate the system, rapid prototyping approach is a very effective way. Consequently, a simulation tool is necessary.

However, in a complex model, the evaluating scope is not limited to just mobility, but including the communication. In order to support wider range of simulation, our research will support the simulation of both mobility and communication and ensures the consistency of integrating support for mobility and communication.

Chapter 2

Background

2.1 Existing Simulators

The simulation of robot and distributed system are not a new study field, currently there are many of related simulators existing. We separated those simulators in to two groups.

First group is robot simulator, most of them provides the simulation of robot movement and sensoring system in 2D or 3D graphic animation. However, their concern is only about robot motion and related factors. Theirs limitation are that, they do not support complex communication of robots, especially the communication for a distributed algorithm. The simulators that are categorized in to this group are for instance, PLAYER/STAGE[2]: Multi-robot and distributed sensor simulator and WEBOTS[3]: 3D realistic-based robot simulation tool.

Second group is network and communication simulator. These simulators allow us to do the simulation of communication on network and help us research in many things such as, message delay, error or network bandwidth. Some of them also provide mobile ad-hoc network simulation feature. However, their original purpose is mainly to be used for network system, not robot. If we want to use them with robot system, we still have to implement many things by ourselves, which is very time-consuming. The example of simulator in this category are, NS2[4]: Network Simulation 2 and Omnet++[5]: A C++-based simulation environment for network system.

We can clearly see that those simulators in both groups are having their own strong points and trade-offs. With this research, we want to present another mobile robot simu-

lator that combined the best of two worlds together. The main purpose of our simulator is to be used for prototyping distributed-algorithm on cooperative autonomous mobile robots.

In the next section, I would like to explain about a simulator called Neko[1], which its framework and its philosophy will be used in our research.

2.2 Neko framework

Neko is a distributed algorithm prototyping framework, which provides many useful features for network and distributed algorithms simulation. Neko integrated with discrete event simulation engine and contains a collection of distributed algorithm protocols, which is ready to be instantly used. Moreover, Neko can simulate the application both on the real network and simulated network. Recently, it was re-factored by considering more about the philosophy of protocol reusability and encapsulation and tends to provide most flexibility for developer. Unfortunately, at the moment Neko provides no support for node movement, which is indispensable for robot system development. Therefore, in order to inherit those abilities of Neko to robots system, we have developed a simulation engine for mobile robots on the top of Neko framework.

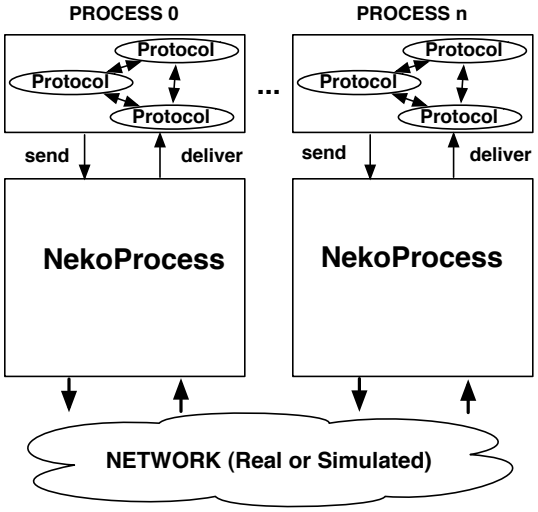

Figure 2.1: Neko's overall model

More concretely say in software level, Neko is a Java programming framework for developing and evaluating distributed protocols using message passing framework. Appli-

cations are connected as the composition of cooperating protocols, connected by a send and deliver operation.

- **Library of protocols**
  An extensive library of distributed algorithm (aka. protocols) has been developed and included within Neko. They are ready to be used in any application. More precisely, these protocols are called microprotocols. The philosophy about them will be talked about in section 3.5.

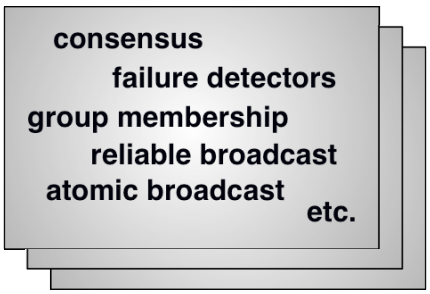

Figure 2.2: Neko's protocols library

- **Real or simulated network**
  Neko supports both simulation on real network and simulated network. As for the simulated, there are many types of simulated network that we can use. For instance, random delay network, metric network or reliable network.

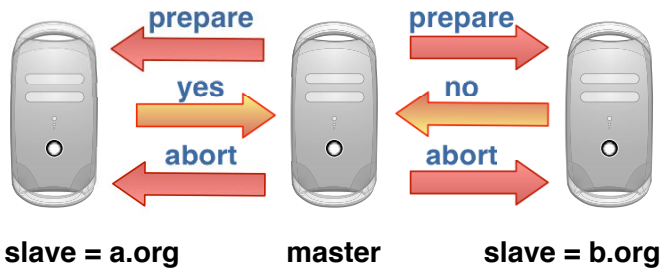

Figure 2.3: An example of simulation running on Neko's real network mode

Chapter 3

Simulator Model

In this chapter, we are going to talk about the specification of simulator model. The figure 3.1 shows overall model after we have integrated mobile robot engine with the original Neko's model. As for robots engine, we have 2 layers which are interface layer and virtual world layer.The interface layer provides control for robots mobility and sensor modules to user.Virtual world layer is the layer of simulated world that robots are existing in. As a result, the integration of mobility and communication can be ensured. We can let robot move in virtual world while having them send/receive messages using Neko.

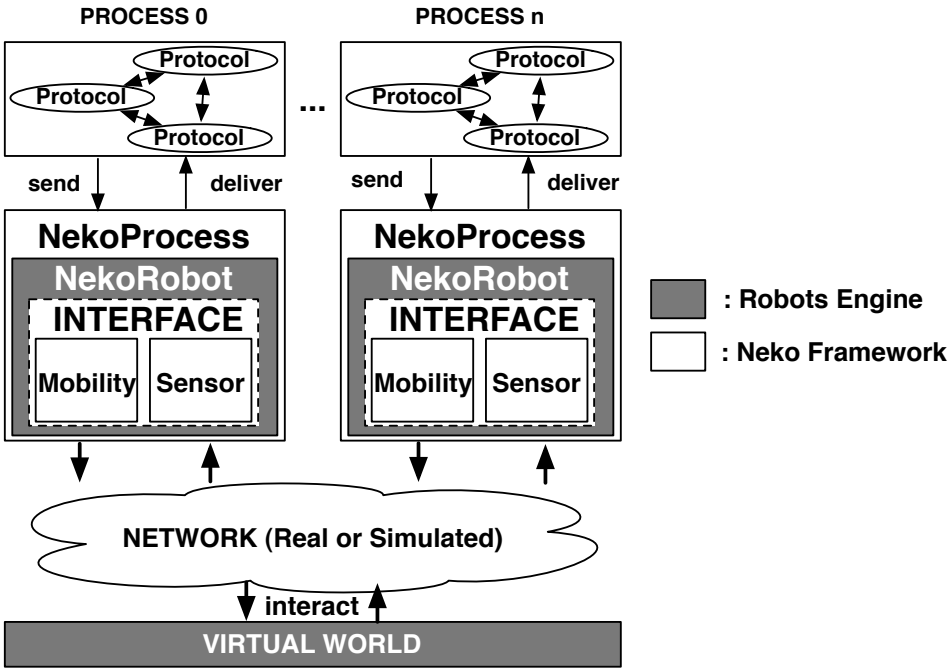

Figure 3.1: Overall Model after integrated robots engine with Neko

Next we will explain about each components in this model. The simulator's model can be divided as the following sections,

3.1 Mobile Robots

Mobile robots are the objects those have the capability to move around in their environment and are not fixed to one physical location as industrial robots, which usually are attached to a fixed surface.

In this simulator, robots are designed based on the real physical robots model. To say specifically, each mobile robot is represented as a the process that logically has its own memory, processing unit and control separately from other robots. In software level, robots contain the following programmatic data informations. For an example of robot definition please refer to figure 3.2.

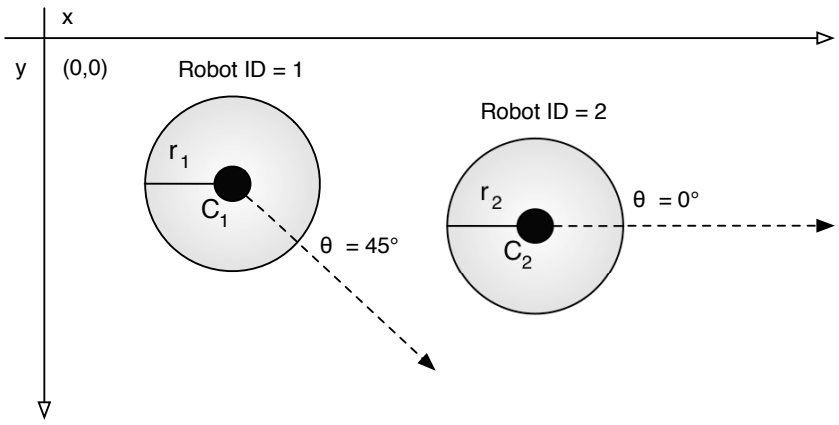

Figure 3.2: Robot id 1 facing to 45 degree and robot id 2 facing to 0 degree

- **ID Number**
  A unique identity number of each robot in the simulation system.

- **Size**
  Size (or to say volume) of the robot, represent by using radius value. Currently this simulator supports only the circular-shaped robot which has same length in both width and height.

- **Angle**
  Angle, is the degree of angle that robot is facing to at a certain time. The angle will be calculated from the origin point (0,0) count anti-clockwise to the target angle.

3.1.1 Motion Modules

Firstly, the most important thing for the mobile robot system is their motions. Robots motions are represented with vector movement. In this simulator, we describe a robots motion as a vector from the their depart coordination to their destination. See figure 3.4 for more understanding.

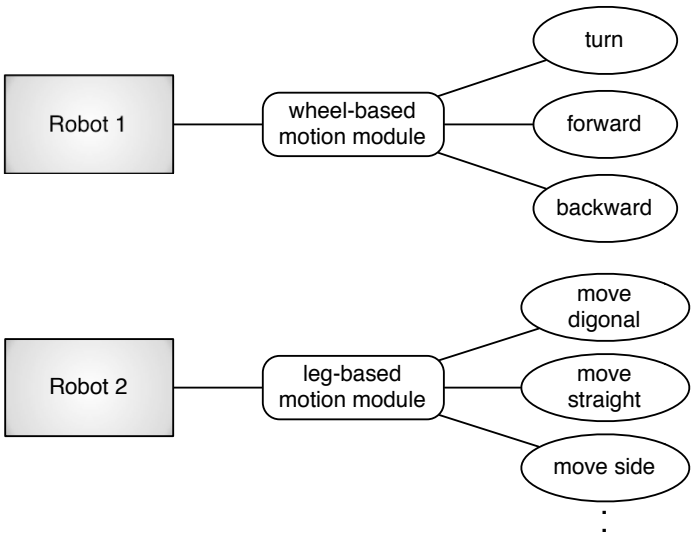

Figure 3.3: An example of varies motion modules.

Motion module is the module that let robot move in the virtual environment. The motion module is also can be called actuator in physical robot. It it is the module that will tell how a robot move. A robot may attached difference motion modules from the others and using totally difference motion patterns. For instance, a wheel-based robot and a spider-shape robot might be attached with difference motion module and move differently. Furthermore, depending on developer,a robot might be attached with multiple motion modules and provides varies motion pattern.

Each motion module consists of one motion pattern or many motion patterns that they are providing. For instance, a wheel-based robot's motion module may provide a turn,

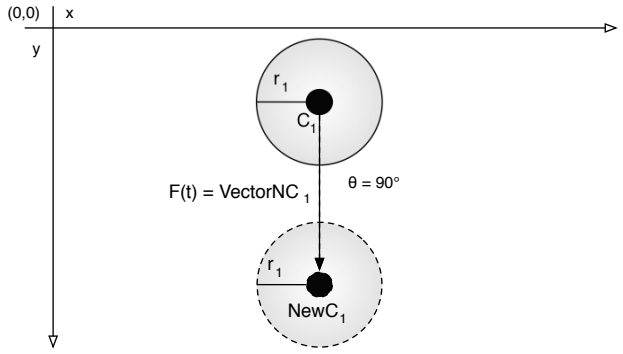

Figure 3.4: A motion vector that was generated by the motion module.

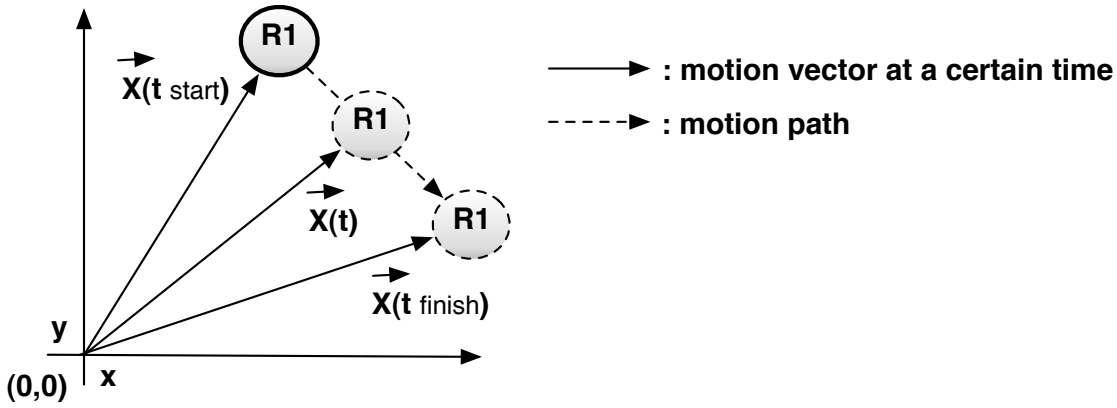

Figure 3.5: An example of motion vector calculation.

forward and backward motion pattern. See figure 3.3 for an instance of motion module.

A motion pattern's main task is vector calculating and generating. To put it simply, it get a motion request from robot, then calculate the vector of motion from that request. Therefore, any motion pattern that can be calculated with the mathematic equation can be created as a motion pattern. Developers are allowed to create their own motion pattern for robots as long as that motion pattern can be described in equation and return the correct motion vector for the robot at a certain time. See figure 3.5 for the vector calculation mechanism in each motion pattern.

Currently, our simulator provides a basic motion module which designed based on the movement of wheel-based robot. A basic motion module (or to say a default motion module) contains three motion patterns, which should sufficient for testing distributed

robots algorithm that concentrate on the interaction between robots or groups of them. However, as previously said, developers are also having the option to develop their own motion pattern in the case that the evaluation of complex motion is necessary. As long as the developers follow the philosophy of the framework, the extended motion patterns can be seamlessly integrated to the this simulator easily.

The calculation of default motion module that was integrated to this simulator can be derived into the following equation,

$$NewCoordination = DepartCoordination + \overrightarrow{Velocity} * t \tag{3.1}$$

Alternatively, we can rewrite this equation into the following short notation term.

$$X_{(t)} = X_{(0)} + \vec{V} * t \tag{3.2}$$

$X_{(t)}$ refers to robot's position at the time t while $X_{(0)}$ represents the position of its position before start moving and $\vec{V}$ is the speed vector that contains direction.

As for the default motion module, it provides these following motion patterns,

- **TURN**
  A motion pattern that let robot turn around itself by using its center point as a turning pin point. The calculation for this motion pattern is nothing than a addition of angles. The motion vector produced by this motion pattern is a unit vector that contains only direction (because when turning robot stay in the same place).

- **FORWARD**
  A motion pattern that allow robot to move to the direction that it is facing to. Since the robot just moving straight with this motion pattern, the angle will be the same as depart angle.

- **MOVETO**
  A motion pattern that combine a turn and a forward motion pattern together. It works with global positioning system by calculating angle that it needed to turn to the destination coordination, then do the turn motion pattern to that target angle,then after the turning is finished, it move forward to the destination coordination.

3.1.2 Sensors

Sensor is a module that let robot ask information of itself, other robots or environment from the system. Robots are separately running from the others. They have no any shared memory or shared processing unit. Therefore it has to request the information of surrounding environment via its sensors. In this simulator we divided sensors into two groups as the following.

- **Pull-model based sensors (a.k.a. Query-based sensors)**

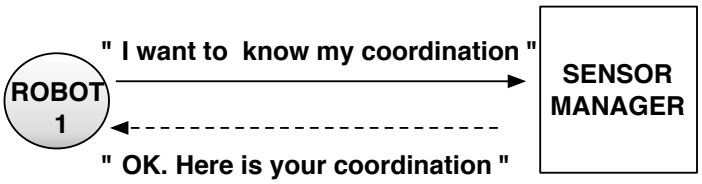

Figure 3.6: An example of pull-model based sensor.

These sensors are the query-based sensors. They will receive a sensor request from robot at a certain time, then wait for the reply from the system and forward that reply after it came to robot. These sensors are called pull-model based sensors. Examples of sensors in this category are gps sensor and view sensor.

- **Push-model based sensors (a.k.a. Notification-based sensors)**

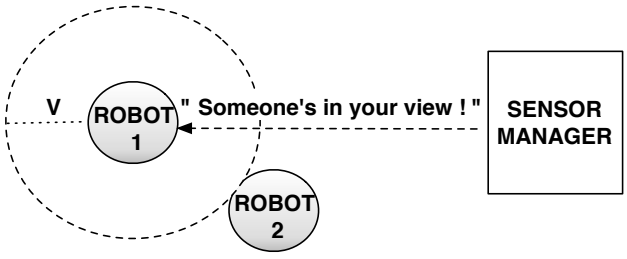

Figure 3.7: An example of push-model based sensor.

These sensors are opposite with above query-based sensors. They will automatically repeat the sensoring mechanism with the time-interval for each sensoring period. They works asynchronously with robots after they have been activated. While they

are re-sensoring with a interval period, if the interrupt condition has become true they will send an interrupt message to robots and give them a sensor's reply message. These sensors are called push-model based sensors. An example of a sensor in the category is a proximity sensor. Proximity sensor will interrupt and send a sensor message to robot if there is other robot try to get into the sensor range (aka. sensor's view size) of this robot.

3.2 Virtual Environment

Virtual environment (also know as virtual world)is the virtual plane-area that is simulated based on the real-world model's space. It is a space that robots are existing and moving in. All of the actions from robots are happening in this simulated world. In this simulator, virtual world based on the Cartesian coordinate system which composed of the x-axis and y-axis. The virtual world might be limited for the size in x-axis and y-axis, or might be an infinity space.

3.3 Collision

Collision is the event that a robot collide with another object and lead to the collision's effect between those objects. Actually in mobile robots system, collision can be categorized into two groups, which are collision between robots and collision between robot and obstacles. However, currently our simulator support only the detection of collision between robots.

This collision happen by the one or more robots moving near to the other until the distance between their's border is become zero, that made robots borders are touching together (also known as collision). We also can separate the collision between robots into two subgroups, which are the collision between moving robots, and the collision between a moving robot and a idling robot. As for the second case, when a robot is not moving and is in idle state, it will become an obstacle for the other robots those are moving in the system.

The other collision that our simulator does not support but should be mentioned is the collision between robot and obstacle. This group of collision is the collision that occur between a robot and an obstacle object in virtual world. As previously said, the virtual world might not be only a simply plane space. It might be a space that consists of many

obstacle objects in its area. The obstacle might be a object that is simulated from the real-world's objects such as a rock, a car or even a wall. Whatever that are, they all do the same role, that is, blocking a moving robot from going further.

In fact, considering real-world model, obstacle object may be very vary and not limited to just the objects those blocking robots way. For instance, a river may not block a robot from moving but just slow its movement down (in the case that robot failure from wetting is not considered). Or another example, a hole that robot can fall into and partly damage, or even completely fall and disappear from the system. However, our simulator's aim is to making the framework as simple as it can in order to reduce the complexity in user's setting. Hence, currently our simulator does not provide support for such a complex obstacle yet.

3.4 Robot's Communications

After we have talked about robots components and their virtual world, in this section we are going to talk about their communication.

As previously described in chapter 2, Neko's original purpose is for rapid prototyping distributed algorithm. In order to simulate the environment for those distributed algorithms, network management ability has a very important role for this field of simulations. In order to support those simulations, Neko provide the ability to allow user to experiment their algorithm on both real network and simulated network. Neko also concentrates on the reusability and encapsulation of communication's protocols which this feature can be considered as Neko's strongest point.

After we have successfully developed the mobility model on the top layer of Neko, we moved to the next phase and integrated it with Neko's communication framework. As a result our simulator now support both robots mobility and their message transmissions.

Now we go back to mobile robot's world. In the cooperative mobile robots simulation, in some models the motioning and sensoring might not enough for evaluating an algorithm because we are not simulating just one robot. We are talking about multiple groups of robots which each group might consists of a very big amount of robots like one thousand

or ten thousand robots. In order to effectively controls those robots and increase the throughput for this robot system, the cooperation between them is an extremely crucial factor.

We have known that the cooperation between them is very importance, but how can they cooperate to each others? There are two methods that they can provides cooperation to each others. First is by using a sensor. A robot might use a sensor to check the situation around them and if necessary it may provide help for other robots. For an instance, a leader robot uses a view sensor to check number of robots in a certain area. If some robots are falling behind it will wait for them. If all robots have already gathered in the destined area, a leader robot will start moving and lead the group to the next target. Other than using sensor, the second method is by using communication which is the main purpose of this section. Since each robots are having their own processing unit and memory, they do not have a shared memory so that they can share information together. In order to share a certain information to other robots, a robot has to do the transmission (aka. communication).

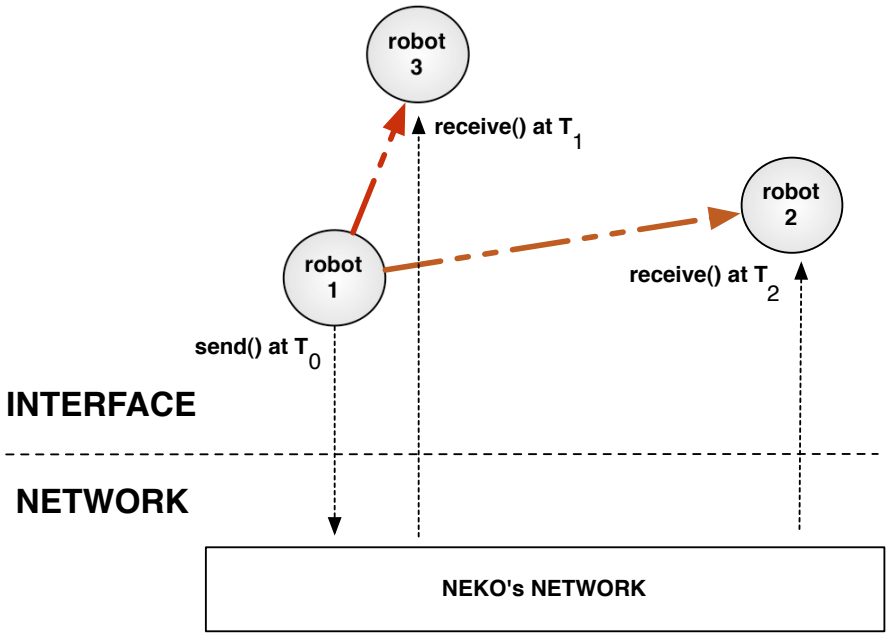

Figure 3.8: An example of a robot broadcasting a message to other two robots

In this simulator, Neko's network framework has been used for simulating communication environment. As we see in the figure 3.8, in the interface layer we can see a robot

broadcasted a message to other robots at time T_0. However, in the back-end mechanism the robot did not send a message to other robots directly. It sent a message to Neko's simulated network (as seen in figure 3.8 as network layer). Then the simulated network will generate delay of message and order of receivers based on the communication algorithms (aka. microprotocols, will be explained in the next section) that have been using. After that, the message will be successfully delivered to the target robots eventually, which is the delivering of messages at T_1 and T_2 in the example in figure 3.8.

3.5 Microprotocol Framework

In this section, we are going to talk about the philosophy of a framework called microprotocol framework. The Neko distributed algorithm prototyping toolkit that we extended the mobility model for, was entirely based on the microprotocol framework's philosophy. Or to say it concretely, Neko was firstly developed on the traditional protocol stacks and was re-factored to completely support microprotocol framework later.

Before getting in to the detail of microprotocol framework, we would like to briefly talk about the traditional protocol stacks. Protocol stacks have been used as a set of collaborating components which these components are called layers. The arrangement of layer's order and communication patterns are fixed, which made each layer can only communicate with the layers directly above and below itself. In order to solve this limitation and provides more flexibility and reusability in framework, the newer framework with new philosophy has been proposed. The framework is called microprotocol framework.

Microprotocol framework, is the framework that permitting the use of finer grained components called microprotocols. Microprotocols are small structuring units that are self-contained and clear in what service that are providing. The basic promise of microprotocol framework is a full separation of concerns between programming microprotocols and composing them. Therefore these two tasks can be carried out by different people with a minimal interaction. This feature can be called multi-level programming. For more understanding see figure 3.9.

From the figure 3.9, the finer grained structuring unit (left side) is called microprotocols. The people who define the definition and develop these microprotocols are called

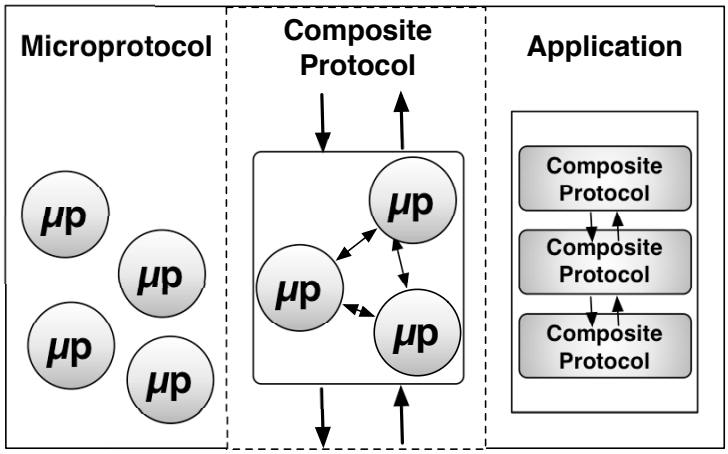

Figure 3.9: Multi-level programing support in microprotocol framework

microprotocols programmers.

Next, the components in the middle part of this figure is called composite protocols. Depends on the author the name calling this type of components might difference. Other developers sometime calls them as complex microprotocols or protocol layers.

Composite protocol is the set of multiple microprotocols and their interconnections. Developers in this programming level are called microprotocols composers. Microprotocol composers are able to plug microprotocols together, without having to know or change the code of the microprotocols.

Last, the final part on the right side is an application. The application consists of multiple composite protocols that are working together while linking by the interconnections between them. Though normally developer in this programming level is usually the same team with microprotocol composer, however in a complex application the application programmers may be assigned to this programming level separating from composers.

In figure 3.10 we created an example of distributed algorithm for communication using microprotocol's philosophy. This application consists of 4 microprotcols that are providing their own services, which are the following units.

- **Atomic Broadcast**
  This protocol module implements atomic broadcast (aka. Total-order broadcast).

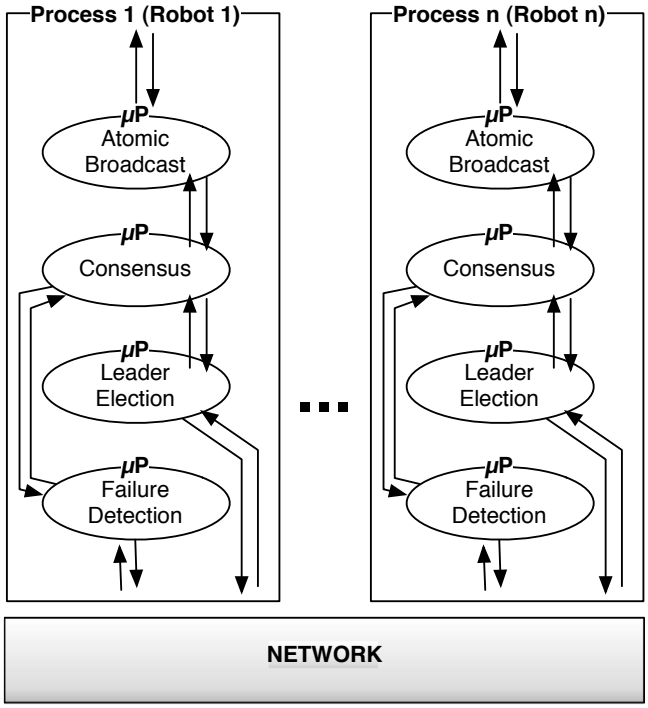

Figure 3.10: An example of application using microprotocol's philosophy

The mechanism is to deliver a message to all receiver nodes same order with sending order. It interacts with consensus module both for solving consensus and failure detection purpose.

- **Consensus**
  This protocol module implements an algorithm for solving consensus problem. It interacts with failure detection module to monitor and detect an unreliable node. It also interacts with leader election in case that the new leader is needed.

- **Leader Election**
  This protocol module suggests a new leader to application. It interacts with consensus to ensure same agreement for all nodes.

- **Failure detection**
  This protocol module implements a failure detection based on network condition. For ex., using ping messages. It interacts with consensus for permitting the same agreement on suspected node.

In previous example of microprotocol-based application, with the advantage of micro-

protocol framework, the composer does not needed to know the programming code in each microprotocols. The composer can easily create a complex application by just plugging microprotocols together without having to change the code in microprotocols.

Now we have talked both mobility model for mobile robots system and communication framework on Neko. As a result, the combination of mobility model and microprotocols-based communication framework of Neko make it possible for developers to create a prototype of complex application of cooperative mobile robots system. We believe that this approach will lead us to more effective way for rapid prototyping cooperative mobile robots application which has the needs for evaluating on both mobility and communications algorithms.

3.6 Discrete Event Simulation Engine

The engine of this simulator is implemented based on the discrete event simulation engine. Discrete event simulation is a simulation system that represented as a chronological sequence of events. Each event occurs at an instant time and make a change of state to the system. Since each event happen in an instance time, the mechanism do not pay attention to the interval between events. Only the event at a certain time will be recorded and marked to the system. Some other reference report also called the time-management system for discrete event simulation engine as time-warp mechanism.

Originally the discrete event simulation's philosophy was suitable for the system that the change of state can be marked to the system as a new event. However, as for mobile robots system, when we want to implement the discrete event simulation system to robots simulation engine, there were several difficulties in implementation which will be spoken in the following part.

Unlike physical-time system that use the sequence time system like the real world's time (see fig.3.11), the discrete event simulation engine's time-warp mechanism cannot provide the information of system's state in the interval period between two events (see fig.3.12). For the system like network simulation system, there might be no problem with marking only message in/out events. But, considering robot system, marking only their depart and arrive event is the same meaning with robot are doing the teleportation (aka. warping).

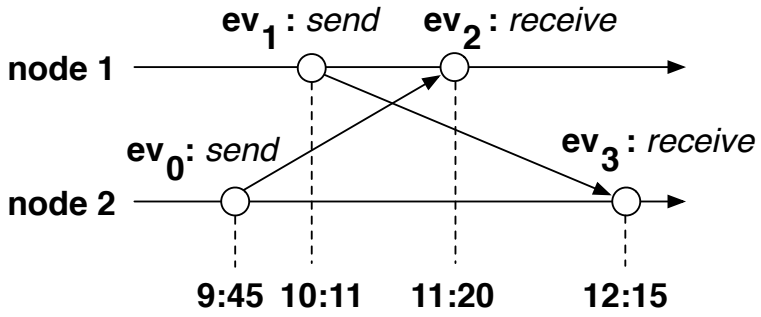

Figure 3.11: An example of physical time system.

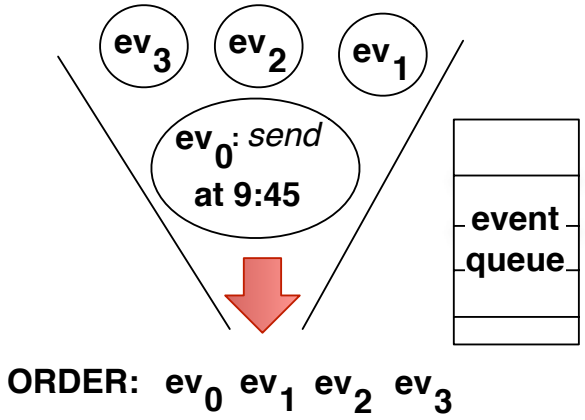

Figure 3.12: An example of virtual time system in discrete event simulation

In mobile robot system, each robot is alway moving on the coordination in virtual space. That means their positions (or to say states in the system) are always changing. Therefore, in some situation the information from the marked depart event and arrive event might be inadequate. For an example, a robot depart from point A at 7th second and will arrive at point B at 10th second. However, what if we want to let the sensor check the position of this robot at 8th second? Since the system only marked the depart and arrive event, how can we know the exact position of this robot at 8th second? See the figure of this example situation at fig.3.13

As for the solution, we can get the information on that interval-period between two events by using vector calculation. As previously said, a robot's motion can be represented as a vector of movement. All vectors of movement in this simulation system can

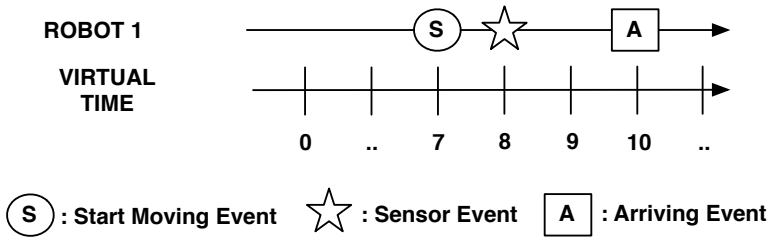

Figure 3.13: A figure shows robot requesting sensor while it is moving

be derived with the equation from equation 3.2 which is $X_{(t)} = X_{(0)} + \vec{V} * t$. While $X_{(t)}$ refers to robot's position at the time t and $X_{(0)}$ represents the position of its position before start moving and $\vec{V}$ is the speed vector that contains direction. As a result, with the above equation we can get exact coordination of a robot at any time t and be able to solve the previous positioning query problem.

However, on the good side, in spite of the lacking of state's information in the interval period between two events, marking robot's motion as a depart and arrive event will dramatically increase the speed of simulation. For example, if we implement the simulation of a group of robots with physical-time based simulator. In the case that robot's speed is extremely slow or the distance to move is very far, the simulation might take a very long time to finish all robots actions. But, with the discrete event simulation engine's virtual time system, the system will jump to the arrive events of all robots in an instant time. As a result, with the discrete simulation engine's mechanism the simulation time can be incredibly shortened.

Chapter 4

Simulator Architecture

The content in this chapter is about the system architecture implemented in this simulator. The explanation starts from overall big-picture of the simulator system then concretely detailed about each component and then an explanation about output from simulation.

4.1 Overall Architecture

The structure consists of three layers. Algorithm layer, interface layer and engine layer. Algorithm layer is the top-level layer which each robot's algorithm is defined and implemented in this layer. To put it simply, in this layer developers only concern about the moving algorithm for each robot. Next layer is the interface layer, the layer that play the role of interface between developer and robot's programmatic codes. Last is the bottom layer, the engine layer. This layer represents back-end core engine of the simulator. All requests received from interface layer are handled here and then processed into simulation output. For more understanding refer to figure 4.1.

The simulator composed of the following core components,

- **Motion Modules**
  The modules that allow robots to move.
- **Sensors**
  The modules that help robots ask information from the system.
- **Managers**
  The modules in the back-end engine that entirely manage their own field of tasks.

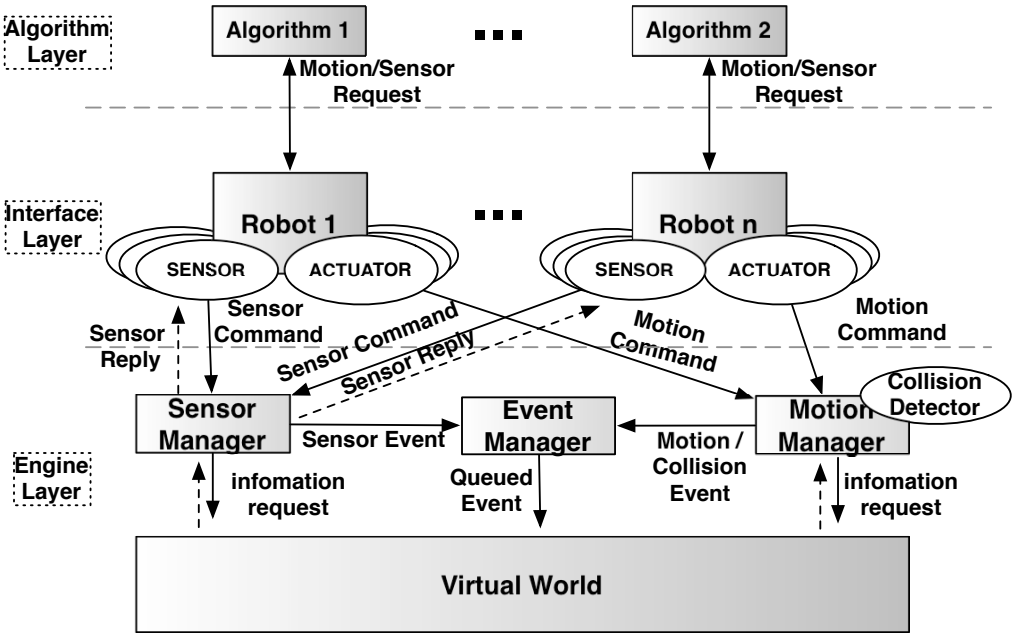

Figure 4.1: Overall structure of simulation system

- **Virtual World**
  The virtual space simulating logical world for all robots.

In the next section we are going to explain these components in the detail.

4.2 Components

4.2.1 Motion Modules

Motion module is a component that allow a robot to move in the virtual world. Since the basic model of this module has already been talked in chapter 3, in this section we are going to explain about the programmatic mechanism.

As been said previously, the simulator is based on the discrete event simulation's philosophy. That means all of the actions from robots will be translated in to records of changes in system's state at a certain time, which these records will be called events.

For robots motions, we translate their motion requests to motion events by using motion manager, then register these events into event queue. When a motion request is translated to motion events, normally two events will be produced. These events are start event and finish event. However, some complex motions may required to produce more than traditional start event and end event. For an instance, a MOVETO motion request (moving robot to the target destination) may requires start-turning event, finish-turning event, start-forwarding event and finish-forwarding event.

Considering more complex motion patterns those might be extended to the simulator in the future, we provide an interface for the motion module which requires user to define two operations (aka. methods in JAVA). The first operation is the calculation part of the motion module, which required to be able to provide the coordination of a robot at any time t. With the event-based simulation system, events will instantly happen at certain time. However, robots are always moving and resulting in the change of their global coordinations. Hence a method to calculate robots position at any target time is necessary for each motion. The returned value is a MotionVector datatype, which is a vector of motion of that robot at time t. Second operation is the returning of array of motion events occur from the motion. As previously said, a complex motion may produces multiple events other than start and finish event, considering extendability, the declaration of this operation is unavoidable. See reference figure at fig.4.2.

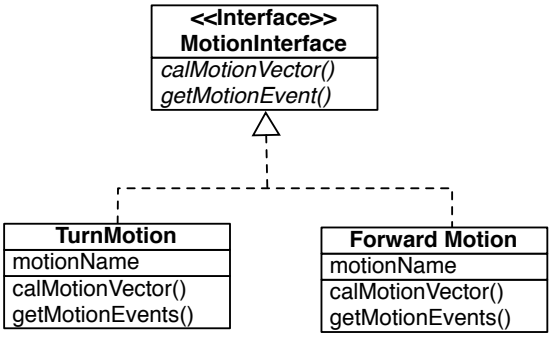

Figure 4.2: Motion modules in UML diagram

TURN Motion

A motion that turn a robot with a target degree in angle. Related calculations are the following,

$$NewAngle = (PreviousAngle + \overrightarrow{TurnVelocity} * t)\%2\pi \tag{4.1}$$

$$NewCoordination = DepartCoordination \tag{4.2}$$

Events happened from this motion : Start-turn and finish-turn event.

FORWARD Motion

A motion that make robot moving forward in to the direction that it is facing to with a certain distance. Related calculations are as the following,

$$NewAngle = PreviousAngle \tag{4.3}$$

$$NewCoordination = DepartCoordination + \overrightarrow{Velocity} * t \tag{4.4}$$

$$\overrightarrow{Velocity}_{X-axis} = MoveSpeed * cos(angle) \tag{4.5}$$

$$\overrightarrow{Velocity}_{Y-axis} = MoveSpeed * sin(angle) \tag{4.6}$$

Events happened from this motion: Start-forward and finish-forward event.

MOVETO Motion

A motion that combined tun motion and forward motion together. Make robot move to a destination point by calculating the degree needed to turn and distance to move, then do the turning and forwarding sequentially.

Events happened from this motion:
Start-turn, finish-turn, start-forward and finish-forward event.

4.2.2 Sensors

The next components that we are going to talk in this section are sensors. Sensor is a module that robots use for asking information from the system (or environment). A robot usually equipped with more than one sensor module. In programmatic software level, a mobile robot may be represented as the collection of multiple sensors which are integrated into a moving object. Without sensors, a robot can not know the information of the outside world.

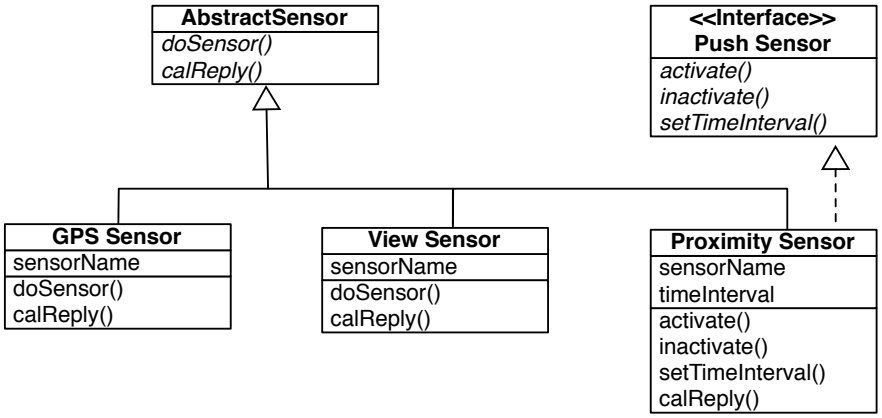

Figure 4.3: Sensor modules in UML diagram

In this simulator we divided sensors into two groups as following.

Pull Model's Sensor

Sensors in this group is called push-model's sensors. They will work using a query-based mechanism, which is sending a sensor request to system and then wait for the reply to return back to the requester robot.

As for the mechanism of pull-sensoring operation in this simulator, first a robot uses its sensor and request for an information of the system. For instance, a position request or surrounding view request. After that, the request will be sent to sensor manager and transform into a sensor event waiting for the time to be triggered. After the sensor event has been triggered, the requested sensor will use informations of the system it get provided by sensor manager, then calculate the result for this request and then return a reply to requester robot as a SensorReply datatype. Since the reply answer's datatype may be difference depending on the sensor's type. For instance, gps sensor might return a Point2D position of requester robot but view sensor will return an array of positions of robots in view range. Hence, the message containing in SensorReply will be stored as a instance of Object class in java. It will be correctly casted into the right datatype before return the reply to the requester robot.

Each sensor will be extended from the provided sensor abstract class. A sensor is needed to define two operations which are doSensor() method and calReply() method. A doSensor() method is the main interface method providing to the robot (or to say to the

user). A robot will ask for a sensor request using this method, then a sensor reply will eventually returned to the robot. Due to the vary types of sensors, parameters required by this method are difference. For instance, a gps sensor may not require any parameter for sensoring operation, but a view sensor requires the range of view in order to perform sensoring operation. Next operation that is needed to be declared is named calReply() method. This method is the back-end calculation part of each sensor. After a sensor event is triggered, this method will be called in order to calculate the reply for a certain request at a event's trigger time. See figure 4.3 for more understanding.

The following sensors are the pull model-based sensors those are provided in the simulator as default modules.

- **GPS sensor**
  A sensor that provide robot's global coordination at request time.
- **View sensor**
  A sensor that return an array of position of robots those are in the range of sensor's view size.
- **Proximity sensor**
  A sensor that return an array of distance to objects in the range of sensor's view size. Note that the proximity sensor can be both pull model-based and push model-based sensor.

Push Model's Sensor

Sensors in this group are called push model's sensors. They are sensors that waiting for an activation, and after they have been activated they will automatically do the sensoring operation with a time interval until they get inactivated or the simulation is finished. The push model-based sensor will not always return the sensor reply to a robot every time they do sensoring, but they will return the sensor reply to a robot only when the interrupt condition has become true. For instance, though a proximity sensor was set to re-sensoring in every 5 second of time interval, it will interrupt a robot and return the sensor reply only when there is a robot come closer into the sensor's view range in order to warn the robot before the collision happen.

Except the re-activating with a time interval mechanism, the main sensoring operation is similar to previously spoken pull model-based sensors. Therefore the push model-based

sensors extends an abstract sensor class as pull model-based sensors do. However, in order to support for the re-activation mechanism, they are required to implements another interface called PushSensorInterface. See figure 4.3 for more understanding.

Below are the push model-based sensors those are provided in the simulator as default modules.

- **Proximity sensor**
  A sensor that will send an interrupting sensor reply to warn a robot if there is another robot come closer into sensor's view range.

- **Contact sensor**
  A sensor that will warn a robot if a robot has touched with other robots or objects.

4.2.3 Managers

After we have known the motion and sensor modules, in this section we are going to explain about the manager classes. These manager hander the requests from robot and manage the system in the back-end. They also provide the collection of many useful methods for the other modules in their management area. Because of the data encapsulation, only the manager-class's components have authority to access to virtual world's informations. The other modules are allowed only to request those information via their manager.

The manager classes in this simulator are as the following.

Motion Manager

Motion manager is the core component that handle entire motion operations for all robots. Motion manager provides many of useful functions those related to robot mobility. It also responsible for the collision detection for the robots. The task of motion manager starts right after there is a motion request sent from robot. It will update the system's state (aka. all robots statuses) to current time before doing any calculation. Then it will analyze the motion request that received from requester robot and transform into motion events. At this point, motion manager have known that there was a change to the system. Therefore it will detect for the collision which may happen from the new motion request. If there will be any collision occur, it will create the collision event and ask event manager to register both the new motion events and the collision event to event queue. If there

was no any collision found it will just register only the new motion events. See figure 4.4 for reference.

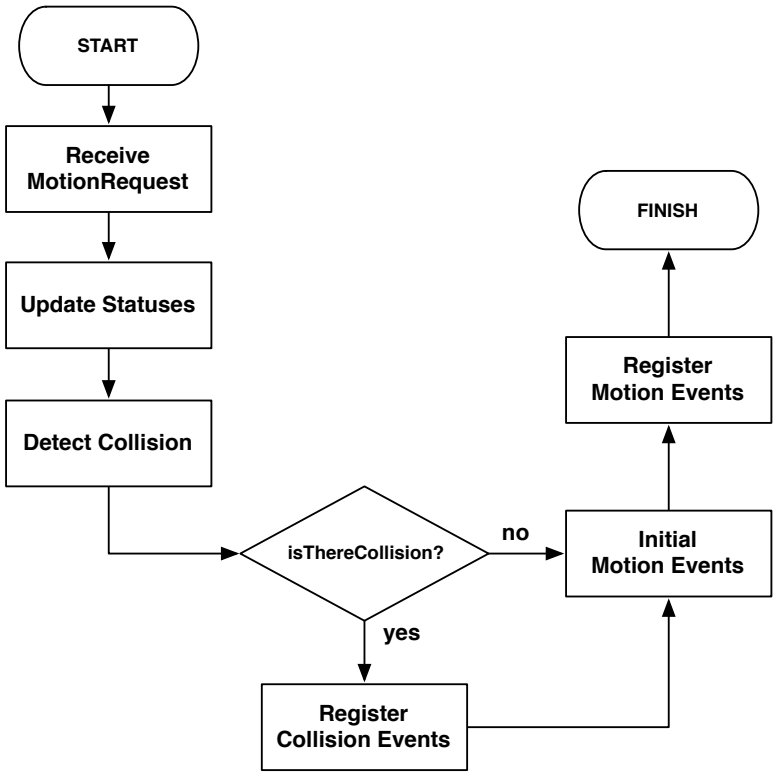

Figure 4.4: Flow chart of the motion manager's working mechanism for one motion request

Sensor Manager

Sensor manager controls most of the main operations relating to sensor system. It starts the work after the sensor request was sent from a robot. Before do any calculation, it update the system's state to current time same way with motion manager do. Then it will create the sensor event from the sensor request it received from the robot and then ask event manager to register this sensor event to event queue. In the same time, it checks whether this sensor request is a push sensor request or pull sensor request. If this request was a request from pull sensor, it will register the new sensor event and the task will finish here. However in the case of push sensor request, an optional task of re-activating sensor is needed. Therefore, if this request is a push sensor request, after the new sensor event has been registered, it will wait for the time interval of the requester push sensor and then repeat the sensor operation again at the next time interval. It will repeat this

mechanism until that push sensor is inactivated or the simulation has finished. See figure 4.5 for reference.

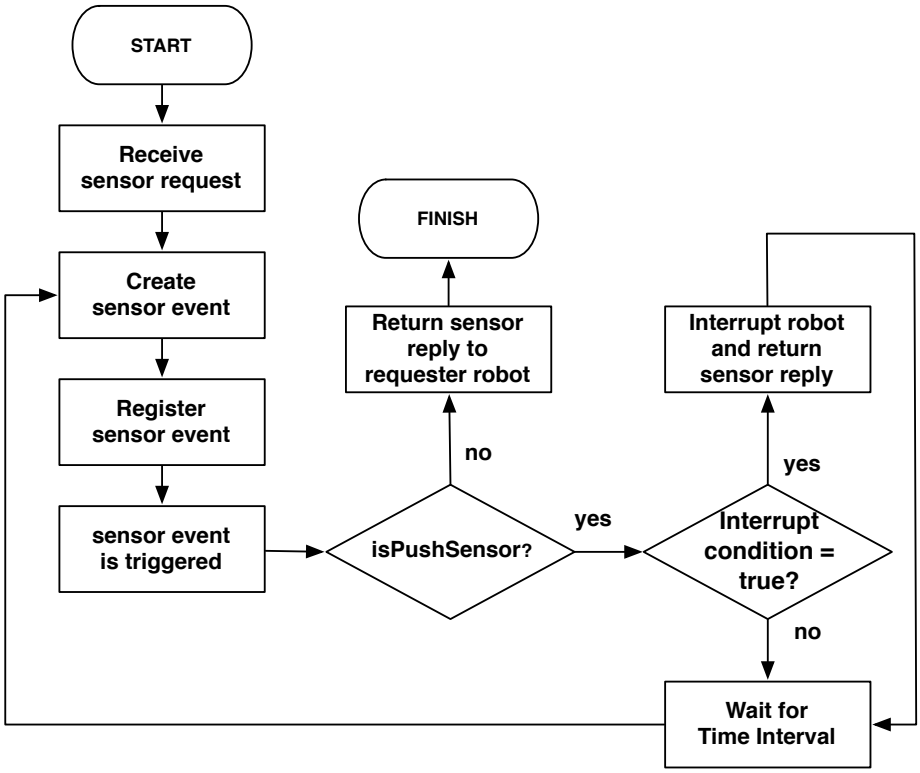

Figure 4.5: Flow chart of the sensor manager's working mechanism

Event Manager

Event manager is the manager that controls all of the task related to events in the system. It has an event queue that stores all of the events of the system. All of the operations related to event management has to be done by the event manager only. The main services that it is providing are event registering and event canceling/removing. It also provides a function to return an array of events stored in the queue for any target robot in the case that developer want to check the future-events of a robot before performing an extra action.

4.2.4 Virtual World

Virtual world is a logical space that robots are existing in. It is a almighty module that knows everything about the robots without having to be told from the other modules. It

is the heart of the simulation engine which storing all of the robots statuses at the current simulation time. The state of each robots will be updated from time to time (specifically say, every time a new event is triggered). The virtual world ensures that the statuses of all robots storing in itself will always up-to-date with the current simulation time. A direct access from an external module to virtual world's database is strictly prohibited due to the data encapsulation. Only the manager-class's components have the authority to access its database.

4.3 Events

All of the state changing (aka. check point) in the system are represented as events. Every time an event is triggered, something happens and mark the change of state to the system. Except for the sensor event that will just ask the information from the system and then return that data to the requester robot. In this section, we are going to explain about the detail of each event.

However, before that we have to know that there are two types of event in this simulator which are eventually-happen event and conditional event.

- **Eventually-Happen Event**
  Events in this group are the traditional event that eventually-happen in the system. The events will be marked with the trigger time. Whatever happen, as long as the simulation has not been terminated, they will certain happen at their trigger time. The events in this group are, motion-start event and sensor event.

- **Conditional Event**
  Events in this group are as its name, in some condition they might not happen in the system and get removed from the event queue. For an instance, a collision event that has been registered to system at t_0 and should happen at t_3, will be canceled if the robot change its motion path before t_3. Because it has changed its path before the collision take place. In contrast, if the collision event happened, the motion-finish event will be canceled instead. In conclusion, events in this groups are motion-finish event and collision event.

4.3.1 Motion Event

A motion event represents the change of a robot's position and its status. It will be marked to the system whenever it get triggered. Below is the flowchart show operations

related to motion events since the time event is created until it finished its task and is removed from the event queue. A motion request from robot will be processed into a motion-start event and a motion-finish event.

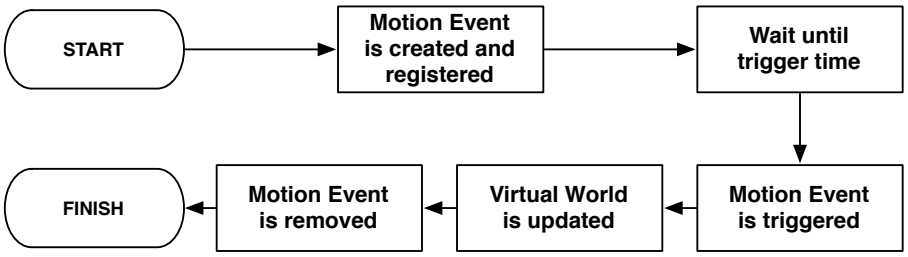

Figure 4.6: Flow chart of the operations related to motion events

4.3.2 Sensor Event

A sensor event is like a request of a snapshot of the system. It will not mark any new state to the system like motion event do, but it will do a snapshot of the system, get information from the system at a certain time, then process this information, transform the information into sensor reply and return to requester robot. Below is the flowchart show operations related to sensor events since the time event is created until it finished its task and is removed from the event queue.

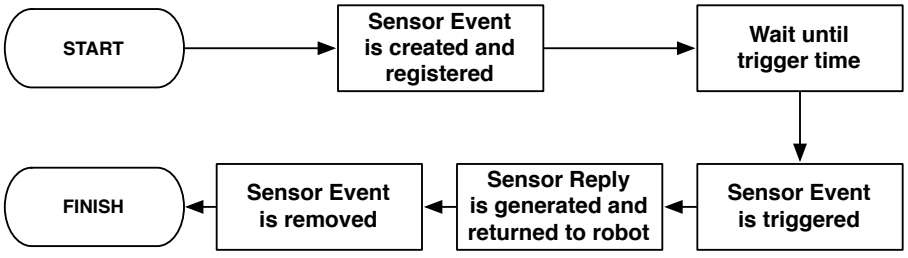

Figure 4.7: Flow chart of the operations related to sensor events

4.3.3 Collision Event

Collision event is a difference from motion event and sensor event. Because collision event is a conditional event. To put it simply, even a collision event has already been registered into the system, depending on the condition of the robot, this collision event might not

happen and get canceled. The condition that make a collision event get canceled is the changing in motion of a robot after the collision event was registered. For instance, robot A and robot B are moving. With current direction and velocity, the collision between them was detected at t_0 and should happen at the time t_2. However, at the time t_1 (which t_1 happen before t_2) robot B has changed it movement and go to a difference coordination. At this point, the previous collision will become invalid and has to be canceled. The motion manager has to do the collision detection again in time t_1. Below is the flowchart show operations related to collision events since the time event is created until it finished its task and is removed from the event queue.

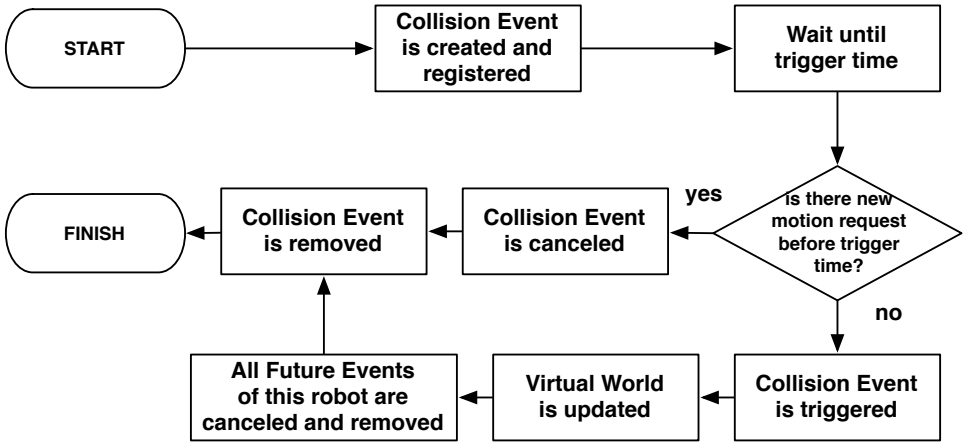

Figure 4.8: Flow chart of the operations related to collision events

4.4 Collision Detection

In this section we are going to explain about the collision detection mechanism using in this simulator. Collision detection operation has a very importance role in motion robot simulation. Because many of robots algorithms have the ultimately goal to effective utilize their limited resource. Therefore, if a collision occurred in system that means we are losing our resource in some ways (robots, energy or even a time). Because of that reason, we aware of the important of the collision detection in the robot simulation system and have integrated this mechanism into our simulator's engine.

Before get into the detail of the mechanism used in our simulator, we would like to talk about the popular mechanism being used for collision detecting. We divided the methods that frequently used for detecting collision in robots simulation into two groups

as following.

- **Detecting using time slice**
  This type of detection mechanism has been using in a number of mobile robot simulators including Player/Stage[2]. It works by slicing the time period that a robot do moving action into many smaller time intervals. Then repeatedly detect for the collision at every time interval. To put it simply, it is the mechanism that put many checkpoints into the system to observe each robot's movement. For instance, a robot requests to do a movement action which will start at $1^{st}second$ and finish in $10^{th}second$. The collision detector knows that this motion will consume $10seconds$ to finish. However, it does not know where and when the collision will occur. Therefore it slices that $10seconds$ period in to many smaller time intervals. Let's assume that it sliced with the time interval of $2second$. Hence, the collision will be checked at $3^{rd}second$, $5^{th}second$, $7^{th}second$ and $9^{th}second$.

Figure 4.9: Before doing the time slicing

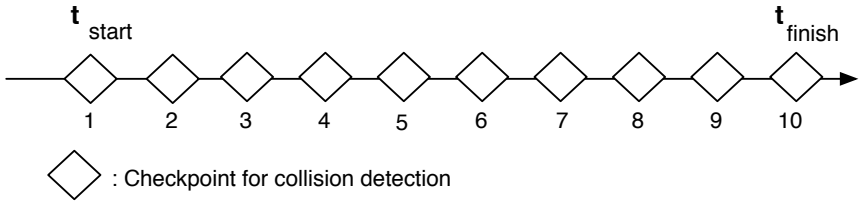

Figure 4.10: After the time slicing have done

This method has both advantages and trade-offs. The good side is, the mechanism itself is easy to implement because the detector does not have to consider about the future-events. It just let the system run and perform the detection at every checkpoints. Furthermore, while using this method we do not have to worry about the shape of robots. They can be any shape because the detector does not have to

predict any collision by using geometrical calculation. It just let the system run and take a checkpoint at every interval.

However, there are several trade-offs in this method as well. First, the number of steps in calculating is incredibly big. Because it needs to calculate at every time-interval, hence the steps number of calculation has became as following,

$$NumberOfSteps = TimeUsage \div IntervalBetweenChecking \tag{4.7}$$

The other drawback for this method is the accuracy in detecting. The bigger time interval between each checkpoint the less accuracy in detecting. For instance, let's use the above example that a robot want to do a movement action from $1^{st}second$ to $10^{th}second$. Let's assume that a collision should happen in $8^{th}second$ if the robot does not change its course. With the time-slice method, if the time interval is equal or less than $1second$, it should be able to successfully detect a collision at $8^{th}second$. However, in the case that interval period is bigger than $1second$, the collision might not be found. Let's demonstrate with $3seconds$ interval period. In this case the detector will do the checking at $4^{th}second$, $7^{th}second$ and $10^{th}second$. As a result, no collision was found. Or it might be found but later than the time that it should happen which is $8^{th}second$. Using extremely small interval period may solve this problem in some case, however the steps number of calculation will incredibly increased and the efficiency of the simulator will dramatically drop.

With above drawbacks of this method, we decided to implements out simulator's collision detector with other method which will be explained in the following part.

- **Detecting using mathematic equation**
  This detection method is the method that uses mathematic calculation. The detector will use current statuses of all robots for detecting the collision that will happen in the future. We knows that a robot's motion can be represented as a motion vector at a certain time. More precisely say, their motion vector at a certain time can be calculated from the equation $X_{(t)} = X_{(0)} + \vec{V} * t$ while $X_{(t)}$ is the position of th robot at time t. Therefore, the detector will use this equation to calculate the collision that will occur in the future. From now we are going to talk about how can it detect for the collision using mathematic equations.

We will simulate a situation that two robots are moving from their start coordinations to the difference destinations. Let's assume that we knows that the collision will eventually happen between these two robots. See figure 4.11 for reference.

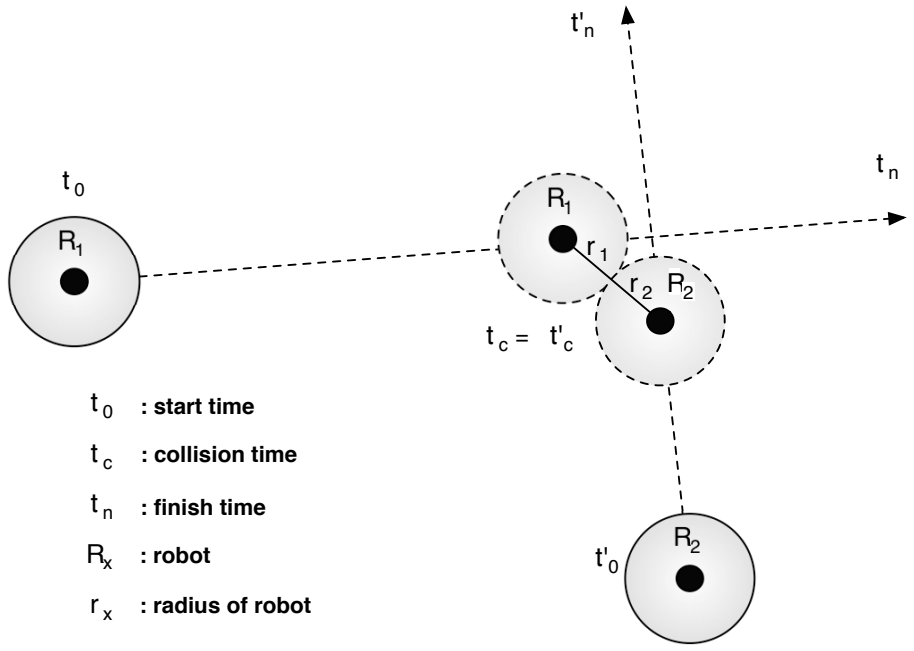

Figure 4.11: An example of a situation that collision will eventually happen

From the figure 4.11, robot 1 (R_1) with the radius size r_1 has started moving at t_0 and will finish this motion at t_n. On the other hand, robot 2 (R_2) with the radius size r_2 has started moving at t'_0 and will finish its moving at t'_n.

First thing that we have to know is, what is the condition that make this two robots collide each other? The answer is, when two robots border are touching together. More precisely say, when the distance from center point of robot 1 to center point of robot 2 is less than the sum of their radius length. This condition can be written as the following.

$$distanceFromCenterPointR_1toR_2 <= r_1 + r_2 \tag{4.8}$$

We knows that the coordination of a robot can be calculated using $X_{(t)} = X_{(0)} + \vec{V} * t$, hence we can transform above condition into the following equation.

$$distance(X_{R_1}(t), X_{R_2}(t)) <= r_1 + r_2 \tag{4.9}$$

Which can be derived into another form as the following.

$$|X_{R_1}(t) - X_{R_2}(t)| <= r_1 + r_2 \tag{4.10}$$

With the above equation we can calculate the collision between these two robots. However, the problem is we do not know time that they will collide together. If we do not know the collision time we can not complete above equation. Therefore, we have to find this missing puzzle first.

After have been considering figure 4.11 carefully, we realize that the collision time for both robots has to be the same. We knows both robots velocity and also their depart coordination, therefore we can find the collision time using these informations which can be derived as following equation.

$$|(X_{R_1}(t_{start}) + \vec{V_{R_1}} * t_{collision}) - (X_{R_2}(t'_{start}) + \vec{V_{R_2}} * t_{collision})| <= r_1 + r_2 \tag{4.11}$$

Now we can solve the equation and get collision time. As a result, after we know collision time, we can calculate the collide position of these two robots as well.

Though this method provide a precise calculation for collision time and collision positions and also consumes only one calculating step for each pair's detection, the trade-off for this method is that it can only be used for robots those are in the sphere-shape.

4.5 Output

In this section, we will explain about the simulation out from our simulator. In this simulator, the main output is a log file in text format. For those who familiar with Neko's logging format, the main pattern of the log is similar with Neko's. With the discrete event simulation engine, each event will mark the change to system and will be recorded as a line in the log file. However, mobile robots simulation is difference from network

simulation. Checking the correctness of each robot's motion with just only log-file is a tough work and consume a lot of time especially when simulating with a great number of mobile robots. Therefore in order to help developer evaluate their application faster and easier, we also provides the animating visualization as an optional choice as well. From now we are going to explain both the log file and animating visualization in details.

4.5.1 Log File

Below are the pattern of the log file produced by our simulator. Each number represents status of the following value orderly. time — R# — pos — des — angle — turn — distance — tSpeed — mSpeed — condition

```
31.059 p3 Event: | 31.059 | 3 | [734.769,518.859] | [734.769,518.859] | 196.621 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
31.059 p3 Event: | 31.059 | 3 | [734.769,518.859] | [423.319,425.888] | 196.621 | 0.000 | 325.031 | 0.000 | 4.570 | Do_MoveTo_Sequence
48.581 p4 Event: | 48.581 | 4 | [371.398,610.116] | [371.398,610.116] | 285.739 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
48.581 p4 Event: | 48.581 | 4 | [371.398,610.116] | [423.319,425.888] | 285.739 | 0.000 | 191.405 | 0.000 | 5.950 | Do_MoveTo_Sequence
53.710 p9 Event: | 53.710 | 9 | [423.319,425.888] | [423.319,425.888] | 199.610 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo
53.710 p9 SensorRequest:  ROBOT# 9 | Positioning with errorFactor 0.0
53.710 p9 MotionRequest:  TIME: 53.710 | ROBOT# 9 Task ID = 1 | MoveTo:  ( 400.452 , 435.529 ) with turnSpeed = 4.761 moveSpeed = 3.978
53.710 p9 Event: | 53.710 | 9 | [423.319,425.888] | [423.319,425.888] | 199.610 | 317.529 | 0.000 | 4.761 | 0.000 | Do_MoveTo_Sequence
59.115 p6 Event: | 59.115 | 6 | [143.396,545.601] | [143.396,545.601] | 336.845 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
59.115 p6 Event: | 59.115 | 6 | [143.396,545.601] | [423.319,425.888] | 336.845 | 0.000 | 304.447 | 0.000 | 4.800 | Do_MoveTo_Sequence
65.838 p5 Event: | 65.838 | 5 | [61.379,510.807] | [61.379,510.807] | 346.796 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
65.838 p5 Event: | 65.838 | 5 | [61.379,510.807] | [423.319,425.888] | 346.796 | 0.000 | 371.769 | 0.000 | 5.943 | Do_MoveTo_Sequence
65.851 p1 Event: | 65.851 | 1 | [379.148,448.388] | [379.148,448.388] | 333.006 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
65.851 p1 Event: | 65.851 | 1 | [379.148,448.388] | [423.319,425.888] | 333.006 | 0.000 | 49.571 | 0.000 | 5.964 | Do_MoveTo_Sequence
74.163 p1 Event: | 74.163 | 1 | [423.319,425.888] | [423.319,425.888] | 333.006 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo
74.163 p1 SensorRequest:  ROBOT# 1 | Positioning with errorFactor 0.0
74.163 p1 MotionRequest:  TIME: 74.163 | ROBOT# 1 Task ID = 1 | MoveTo:  ( 392.737 , 431.771 ) with turnSpeed = 5.774 moveSpeed = 4.389
74.163 p1 Event: | 74.163 | 1 | [423.319,425.888] | [423.319,425.888] | 333.006 | 196.105 | 0.000 | 5.774 | 0.000 | Do_MoveTo_Sequence
```

Figure 4.12: A path of log file from gathering algorithm

This example log is a log from gathering algorithm (refers to section 5.2). There was no any communication between them. There were only the sensor requesting and robots moving actions. As we can see in the header of the log file. The values from left to right are time, robot id, current position, destination, current angle, degree to turn, distance left to move, turn speed, move speed and condition. There also are some another patterns that do not appear in this log such as message sending, receiving and push-model sensor event.

4.5.2 Animating Visualization

The next is the animating visualization. The example of animation can be found in below pictures. These figures are the animation from the gathering algorithm. The first figure (fig.4.13) happen before and second figure (fig.4.14). Though it is difficult to show th motion with captured

screen, we can see that most of the robots are trying to move to the centroid point between their coordinations. The animating visualization is based and produced from the log file. It will read each checkpoint of robots and then generate into the waypoints of moving.

Each robot will be labeled with their id number. Their sizes in the animation panel are depends on the length of their radius. Also several settings for the animation environment such as framerate or panel size are also available. We will talk about this in section 6.1.

Though animating visualization is definitely useful in many ways such as education, presentation or even in evaluation. However, the calculating for animation consumes a lot of resource and time. Displaying animation means the system has to emulate itself into time-slicing system and lead to the dropping in calculating performance. It also ageists the advantages of discrete event simulation system's philosophy. Therefore, in the complex simulation, we recommend to turn off the visualization feature.

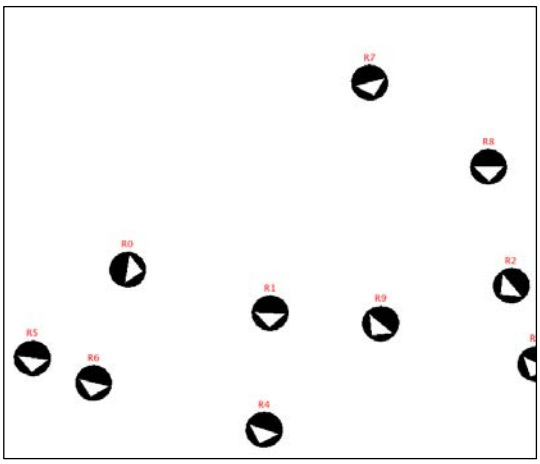

Figure 4.13: An animation output from the gathering algorithm (Before)

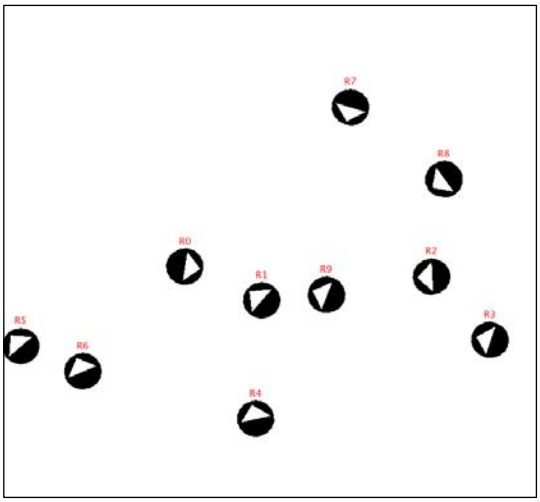

Figure 4.14: An animation output from the gathering algorithm (After)

Chapter 5

Example Simulations

In this chapter, we will demonstrate three examples of the simulations which consists of Random movement, autonomous gathering formation and using communication to ask for help. Source code for each application also included and can be found right after the algorithm explanation.

5.1 Random Movement

The first application is the most simple one. As its name, in this algorithm robots will random new destination and then move to this destination. After they have reached at their destination point, they will repeat the same step, which is random a new destination and move there. They will doing this pattern repeatedly until they have reach the finish condition which is the time limit or finish condition.

This is an algorithm for this application

```
while(currentTime <= LimitedTime){
        Point2D destination = randomDestination();
        double moveSpeed = randomMoveSpeed();
        double turnSpeed = randomTurnSpeed();
        moveTo(destination, turnSpeed, moveSpeed);
}
```

Below is the visual output of this application, a part of log file and source code. The application was tested by using 100 robots run with random move speed and turn speed.

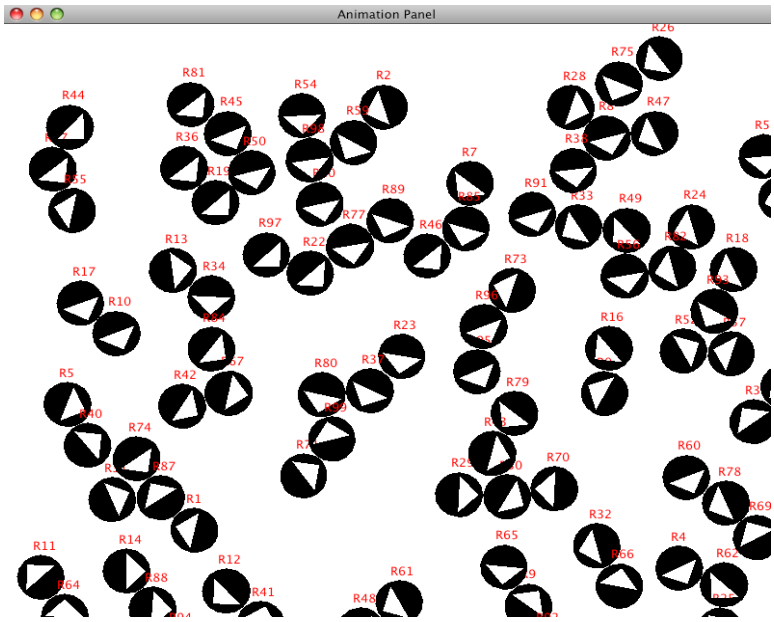

Figure 5.1: An animation output from the random movement algorithm

```
CollisionEvent: | 60.269 | 86 | [753.817,734.028] | [675.772,448.298] | 254.723 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
CollisionEvent: | 60.324 | 52 | [681.815,336.437] | [766.488,290.472] | 331.504 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
CollisionEvent: | 61.029 | 76 | [62.638,749.125] | [400.578,690.181] | 350.106 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
Event: | 62.840 | 28 | [417.171,22.293] | [417.171,22.293] | 263.158 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo
MotionRequest:  TIME: 62.840 | ROBOT# 28 Task ID = 1 | MoveTo:  ( 759.919 , 127.097 ) with turnSpeed = 8.784 moveSpeed = 7.687
Event: | 62.840 | 28 | [417.171,22.293] | [417.171,22.293] | 263.158 | 113.844 | 0.000 | 8.784 | 0.000 | Do_MoveTo_Sequence
Event: | 64.504 | 73 | [559.749,285.555] | [559.749,285.555] | 196.482 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
Event: | 64.504 | 73 | [559.749,285.555] | [70.128,140.692] | 196.482 | 0.000 | 510.601 | 0.000 | 5.245 | Do_MoveTo_Sequence
CollisionEvent: | 65.076 | 68 | [308.290,752.723] | [466.289,699.307] | 341.321 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
CollisionEvent: | 69.726 | 55 | [47.713,181.113] | [41.020,179.761] | 191.417 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
CollisionEvent: | 75.573 | 73 | [504.075,269.083] | [70.128,140.692] | 196.482 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
Event: | 75.800 | 28 | [417.171,22.293] | [417.171,22.293] | 17.002 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
Event: | 75.800 | 28 | [417.171,22.293] | [759.919,127.097] | 17.002 | 0.000 | 358.413 | 0.000 | 7.687 | Do_MoveTo_Sequence
Event: | 80.947 | 40 | [24.364,472.005] | [24.364,472.005] | 321.443 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
Event: | 80.947 | 40 | [24.364,472.005] | [411.861,163.148] | 321.443 | 0.000 | 495.526 | 0.000 | 10.115 | Do_MoveTo_Sequence
CollisionEvent: | 85.940 | 40 | [63.864,440.521] | [411.861,163.148] | 321.443 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
CollisionEvent: | 95.948 | 28 | [565.279,67.580] | [759.919,127.097] | 17.002 | 0.000 | 0.000 | 0.000 | 0.000 | Do_MoveTo_Sequence(Collision)
. Event: | 106.570 | 71 | [259.503,494.880] | [259.503,494.880] | 324.081 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
```

Figure 5.2: A log file output from the random movement algorithm

```
package lse.neko.robotsim.algorithm;

import java.awt.geom.Point2D;

import lse.neko.robotsim.robot.NekoRobot;

/**
 * A simple random algorithm. Speed and destination of robots will be randomed.
 *
```

```
 * SETTING: 10-20 robots with random bornpoint is recommended
 *
 * @author Smath
 */
public class RandomMovement extends AbstractAlgorithm {

    public RandomMovement(NekoRobot robot) {
        super(robot);
    }

    protected void doMotion() {

        double moveSpeed = 10;
        double turnSpeed = 30;

        //Keep the minimum speed at a certain value.
        //Otherwise, if the speed is randomed and become like 0.00001
        //it will take very long time to run animation.
        double minSpeed = 3;
        //random movespeed and turnspeed with the lower bound
        moveSpeed = moveSpeed * Math.random() + minSpeed;
        turnSpeed = moveSpeed * Math.random() + minSpeed;
        //random destination
        Point2D.Double destination = new Point2D.Double();
        destination.x = 800 * Math.random();
        destination.y = 800 * Math.random();
        //implements the motion
        actuator.moveTo(destination, moveSpeed, turnSpeed);
    }
}
```

5.2 Autonomous Gathering Formation

The second application has been briefly said in the previous chapter, it is called gathering formation. To put it simply, it is a algorithm that try to make all robots gather at the closest position as much as they can. If robots have no volume and represented by a point, this algorithm will take a finite time until all robots are close enough and completely be the same position, which

we might reversely say that they will not become one gathering point in an infinite time. Why not? Because they are alway moving. That made sensoring phase of each robot happens in the difference timing. Consequently the centroid point will always changing from time to time. However in the realistic world, robots do have volume (size). The more they com closer, the high possibility collision will occur. Therefore when we implemented this application, we will enable the ghost mode (refer section 6.1) to ignore all collisions.

Below is the algorithm for this application. An almighty sensor that can tell all robots coordinations is required in this application. We used the gps sensor to get all those positions (we can use view sensor with very-wide sensor range as well).

```
while(currentTime <= LimitedTime){
        Point2D[] allpos = gps.getAllPos();
        Point2D centroid = getCentroid(allpos);
        double moveSpeed = randomMoveSpeed();
        double turnSpeed = randomTurnSpeed();
        moveTo(centroid, turnSpeed, moveSpeed);
}
```

In this example, we used 10 robots with ghostmode enabled (means no collision occur). The followings are visual output, a part of log file and source code orderly.

Figure 5.3: An animation output from the gathering algorithm

```
Event: | 80.433 | 8 | [419.290,386.045] | [419.290,386.045] | 35.148 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo
SensorRequest:  ROBOT# 8 | Positioning with errorFactor 0.0
MotionRequest:  TIME: 80.433 | ROBOT# 8 Task ID = 1 | MoveTo:  ( 407.641 , 411.715 ) with turnSpeed = 6.413 moveSpeed = 3.517
Event: | 80.433 | 8 | [419.290,386.045] | [419.290,386.045] | 35.148 | 79.260 | 0.000 | 6.413 | 0.000 | Do_MoveTo_Sequence
Event: | 85.831 | 5 | [401.356,373.418] | [401.356,373.418] | 64.279 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
Event: | 85.831 | 5 | [401.356,373.418] | [417.676,407.296] | 64.279 | 0.000 | 37.604 | 0.000 | 5.452 | Do_MoveTo_Sequence
Event: | 92.728 | 5 | [417.676,407.296] | [417.676,407.296] | 64.279 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo
SensorRequest:  ROBOT# 5 | Positioning with errorFactor 0.0
MotionRequest:  TIME: 92.728 | ROBOT# 5 Task ID = 1 | MoveTo:  ( 407.694 , 413.559 ) with turnSpeed = 7.943 moveSpeed = 5.302
Event: | 92.728 | 5 | [417.676,407.296] | [417.676,407.296] | 64.279 | 83.616 | 0.000 | 7.943 | 0.000 | Do_MoveTo_Sequence
Event: | 92.793 | 8 | [419.290,386.045] | [419.290,386.045] | 114.408 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
Event: | 92.793 | 8 | [419.290,386.045] | [407.641,411.715] | 114.408 | 0.000 | 28.190 | 0.000 | 3.517 | Do_MoveTo_Sequence
Event: | 93.673 | 4 | [417.046,410.886] | [417.046,410.886] | 167.120 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo_Sequence
Event: | 93.673 | 4 | [417.046,410.886] | [409.272,412.664] | 167.120 | 0.000 | 7.974 | 0.000 | 3.106 | Do_MoveTo_Sequence
Event: | 96.240 | 4 | [409.272,412.664] | [409.272,412.664] | 167.120 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo
SensorRequest:  ROBOT# 4 | Positioning with errorFactor 0.0
MotionRequest:  TIME: 96.240 | ROBOT# 4 Task ID = 1 | MoveTo:  ( 404.559 , 411.839 ) with turnSpeed = 5.875 moveSpeed = 5.003
Event: | 96.240 | 4 | [409.272,412.664] | [409.272,412.664] | 167.120 | 22.803 | 0.000 | 5.875 | 0.000 | Do_MoveTo_Sequence
Event: | 99.030 | 2 | [401.356,373.418] | [401.356,373.418] | 106.412 | 0.000 | 0.000 | 0.000 | 0.000 | Finish_MoveTo
SensorRequest:  ROBOT# 2 | Positioning with errorFactor 0.0
```

Figure 5.4: A log file output from the gathering algorithm

```
package lse.neko.robotsim.algorithm;

import java.awt.geom.Point2D;

import lse.neko.robotsim.robot.NekoRobot;

/**
 * In this algorithm robots will use gps sensor to get position of the other robots
 * then calculate controid between them and move to that centroid point.
 * They will do this mechanism repeatedly until reach stop condition.
 *
 * SETTING: more than 3-5 robots with random bornpoint and ghostmode on
 * (Otherwise, they will collide with other robots very fast.)
 *
 * @author Smath
 */
public class Gathering extends AbstractAlgorithm {

    public Gathering(NekoRobot robot) {
        super(robot);
    }

    protected void doMotion() {
```

```
        double moveSpeed = 3;
        double turnSpeed = 30;

        Point2D.Double[] allPos = gps.getAllPosition(0);
        Point2D.Double centroid = getCentroid(allPos);

        double minSpeed = 3;
        moveSpeed = moveSpeed * Math.random() + minSpeed;
        turnSpeed = moveSpeed * Math.random() + minSpeed;
        actuator.moveTo(centroid, moveSpeed, turnSpeed);

    }

    /**
     * return centroid point between all robots coordinations.
     */
    private Point2D.Double getCentroid(Point2D.Double[] allPos) {

        double sumx = 0;
        double sumy = 0;
        for (int i = 0; i < allPos.length; i++) {
            sumx = sumx + allPos[i].x;
            sumy = sumy + allPos[i].y;
        }

        double x = sumx / allPos.length;
        double y = sumy / allPos.length;

        return new Point2D.Double(x, y);

    }
}
```

5.3 Using Communication to Ask for Help

The last example application is an algorithm that using the communication between robots and let them remotely ask the other robots to do some task. In this application, the task is a

motion to go to desired coordination. Since our purpose for this application is to ensure the consistency of message sending/receiving when using simultaneously with robots mobility. As for the network, we used Neko's simulated random network with random lamda equals 10. Since we want to get rid of complexities, we did not consider mutual exclusion in this application. In this application, robots will not move on their own. They will wait for a order from other robot telling them where to go. When robots send orders to the others, the messages will be sent using multicast mechanism (from multiple sender robots to multiples receiver robots). However, when robots receive a message, they will not consider about order of the sending or messages stocked in the queue. They will pick the fist message that arrive to them and then do the task requested from that message.

The following is the algorithm from this application.

```
while(currentTime <= LimitedTime){
        //requesting phase
        Point2D goto = randomPoisition();
        send(gotoMessage) to all robots except myself;

        //waiting phase
        wait for an order from somebody;
        order = receive();
        Point2D reqPossition = order.getContent();
        double moveSpeed = randomMoveSpeed();
        double turnSpeed = randomTurnSpeed();
        moveTo(reqPosition, turnSpeed, moveSpeed);
}
```

In this example, we used 10 robots implemented with Neko's simulated random network. The followings are visual output, a part of log file and source code orderly.

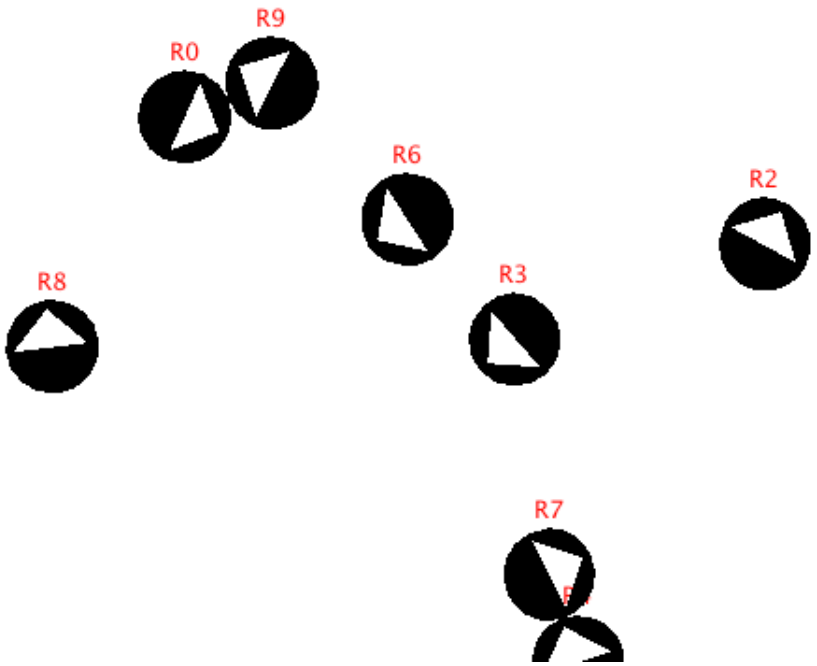

Figure 5.5: An animation output from the using communication to ask help algorithm

```
package lse.neko.robotsim.algorithm;

import lse.neko.robotsim.robot.NekoRobot;
import lse.neko.NekoMessage;
import lse.neko.MessageTypes;

import java.util.Random;
import java.awt.geom.Point2D;

/**
 * In this algorithm, we will test the correctness of message sending/receiving.
 * After have been initialized, all robots will random a destination.
 * Then they will ask the other robots to go there by sending a multicast message.
 * After the message was sent, they will try to receive a message that come first.
 * Then, they will see the content of that message and go to that requested destina
 * NOTE: This algorithm used Neko's active receiver mechanism.
 *
```

```
0.000 p0 messages e s p0 p1,p2,p3,p4,p5,p6,p7,p8,p9 GOTO Point2D.Double[172.6449505651384, 157.88363674423255]
0.000 p1 messages e s p1 p0,p2,p3,p4,p5,p6,p7,p8,p9 GOTO Point2D.Double[123.53058830389317, 181.49180833660705]
0.000 p2 messages e s p2 p0,p1,p3,p4,p5,p6,p7,p8,p9 GOTO Point2D.Double[695.501030467148, 435.213328006639]
0.000 p3 messages e s p3 p0,p1,p2,p4,p5,p6,p7,p8,p9 GOTO Point2D.Double[328.11834379063725, 154.83896287383712]
0.000 p4 messages e s p4 p0,p1,p2,p3,p5,p6,p7,p8,p9 GOTO Point2D.Double[377.58573590119045, 486.4444384384253]
0.000 p5 messages e s p5 p0,p1,p2,p3,p4,p6,p7,p8,p9 GOTO Point2D.Double[392.3105851072991, 358.1539584410355]
0.000 p6 messages e s p6 p0,p1,p2,p3,p4,p5,p7,p8,p9 GOTO Point2D.Double[673.2964725456461, 211.92816013281498]
0.000 p7 messages e s p7 p0,p1,p2,p3,p4,p5,p6,p8,p9 GOTO Point2D.Double[514.6543141475844, 93.14457999542948]
0.000 p8 messages e s p8 p0,p1,p2,p3,p4,p5,p6,p7,p9 GOTO Point2D.Double[227.46880040554026, 272.8846585260699]
0.000 p9 messages e s p9 p0,p1,p2,p3,p4,p5,p6,p7,p8 GOTO Point2D.Double[341.6929815272915, 255.80954521646214]
0.031 p1 messages e r p6 p0,p1,p2,p3,p4,p5,p7,p8,p9 GOTO Point2D.Double[673.2964725456461, 211.92816013281498]
0.031 p1 Algorithm:  Someone told me to go to: (673.2964725456461 , 211.92816013281498)
0.031 p1 MotionRequest:  TIME: 0.031 | ROBOT# 1 Task ID = 1 | MoveTo:  ( 673.296 , 211.928 ) with turnSpeed = 11.000 moveSpeed = 6.000
0.031 p1 Event: | 0.031 | 1 | [431.877,738.603] | [431.877,738.603] | 0.000 | 294.626 | 0.000 | 11.000 | 0.000 | Do_MoveTo_Sequence
0.061 p5 messages e r p3 p0,p1,p2,p4,p5,p6,p7,p8,p9 GOTO Point2D.Double[328.11834379063725, 154.83896287383712]
0.061 p5 Algorithm:  Someone told me to go to: (328.11834379063725 , 154.83896287383712)
0.061 p5 MotionRequest:  TIME: 0.061 | ROBOT# 5 Task ID = 1 | MoveTo:  ( 328.118 , 154.839 ) with turnSpeed = 26.000 moveSpeed = 3.000
0.061 p5 Event: | 0.061 | 5 | [365.926,11.792] | [365.926,11.792] | 0.000 | 104.805 | 0.000 | 26.000 | 0.000 | Do_MoveTo_Sequence
0.101 p7 messages e r p2 p0,p1,p3,p4,p5,p6,p7,p8,p9 GOTO Point2D.Double[695.501030467148, 435.213328006639]
0.101 p7 Algorithm:  Someone told me to go to: (695.501030467148 , 435.213328006639)
0.101 p7 MotionRequest:  TIME: 0.101 | ROBOT# 7 Task ID = 1 | MoveTo:  ( 695.501 , 435.213 ) with turnSpeed = 23.000 moveSpeed = 8.000
0.101 p7 Event: | 0.101 | 7 | [273.727,645.031] | [273.727,645.031] | 0.000 | 333.551 | 0.000 | 23.000 | 0.000 | Do_MoveTo_Sequence
0.103 p6 messages e r p5 p0,p1,p2,p3,p4,p6,p7,p8,p9 GOTO Point2D.Double[392.3105851072991, 358.1539584410355]
0.103 p6 Algorithm:  Someone told me to go to: (392.3105851072991 , 358.1539584410355)
```

Figure 5.6: A log file output from the using communication to ask help algorithm

```
 * SETTING: multiple robots (more than two is suggested) with random bornpoint.
 *
 * @author Smath
 */
public class TestCommunication extends AbstractAlgorithm {

    // message types used by this algorithm
    private static final int GOTO = 1225;
    // registering the message types and associating names with the types.
    static {
        MessageTypes.instance().register(GOTO, "GOTO");
    }

    public TestCommunication(NekoRobot robot) {
        super(robot);
    }

    public void doMotion() {

        double moveSpeed = 10;
        double turnSpeed = 30;

        int n = robot.getN();
        int me = robot.getID();
```

```
        int[] allButMe = new int[n - 1];
        for (int i = 0; i < n - 1; i++) {
            allButMe[i] = (i < me) ? i : i + 1;
        }

        //random destination and ask someone to go there
        Point2D.Double des = new Point2D.Double(Math.random() * 700,
                                                Math.random() * 700);
        NekoMessage msg = new NekoMessage(allButMe, getId(), des, GOTO);
        send(msg);

        Random generator = new Random();

        //Keep the minimum speed at a certain value.
        //Otherwise, if the speed is randomed and become like 0.00001
        //it will take very long time to run animation.
        double minSpeed = 3;

        turnSpeed = generator.nextInt((int) turnSpeed) + minSpeed * 2;
        moveSpeed = generator.nextInt((int) moveSpeed) + minSpeed;

        //pick the first message in queue
        NekoMessage recvMsg = receive();
        Point2D.Double goTo = (Point2D.Double) recvMsg.getContent();
        logger.info("Someone told me to go to: (" + goTo.x + " ," + goTo.y + ")");
        actuator.moveTo(goTo, moveSpeed, turnSpeed);
    }
}
```

Chapter 6

Developer Guide

In this section, we will talk about how to use this simulator. The manual covers the setting of the system, initialization of robots and available motion and sensor commands list. As for the example of algorithm files, please refer to chapter 5. Entire source code of algorithm files have been included in those example simulations.

6.1 Configuration File

Before have the simulation run, first thing that has to do is setting multiple values of the environment. Setting of the environment can be done through the setting file called $robotsim.config$. Below is the available setting parameters.

- **Number of robots**
  Edit the n variable in the following line. n has to be an integer that is greater than zero.
  $process.num = n$

- **Robot's size**
  Size (or to say volume) of the robot, represent by using radius value. Edit the r variable in the following line. r has to be an integer that is greater than zero.
  $robot.size = r$

- **Born point of robots**
  We can specific the born point position for each robot. Optionally, we can also let the system random their born points as well. If the random born point mode has been enabled, the system will random born points for all robot while ensuring their non-duplicated born points area (aka. intersection free). To put it simply, with the random mode on, system guarantees that they will not collide other robots when they are born. With the random mode disabled, user has to define all born points for robots.

$bornpoint.random = true/false$
if above value = false, the following lines has to be defined. x,y can be both double and integer value while n is number of the robots.
$bornpoint1 = x, y$
.
.
$bornpointn = x', y'$

- **Initializer path**
  Robots initializer class file. This class also initialize the protocol stack of each robot. Normally the path of this class should not be changed unless you are familiar with Neko's initializer mechanism. Moreover, output log file's path has to be correctly defined.
  $process.initilaizer = lse.neko.robotsim.RobotSimInitializer$

- **Animation setting**
  Settings related to animation output. Animation can be enabled or disabled.
  $animation = true/false$
  $animation.framerate = frame(s)/second(simulationTime)$
  $animation.panel.width = positiveInteger/Double$
  $animation.panel.height = positiveInteger/Double$
  $animation.logpath = /YOURLOGPATH/yourlogfile.log$

- **Ghost mode**
  With the ghost mode enabled, the will be no any collision occur in the system. It is useful in some algorithms such as gathering algorithm.
  $ghost_mode = true/false$

6.2 Robots Initializer

The initializer is the class that will initial robot's algrithms and protocol stack of each robots for their communication. For more information on how to compose communication protocols, please refer to Microprotocol section in 3.5 and Neko's documents. Breifly, a coomunication protocols is connecting to other protocols with sending and receiving operation. Below is an example of how to initialize robots algorithm and communication protocol stack. In this example, Total-ordered broadcast protocol has been composed with Neko's simulated network.

```
package lse.neko.robotsim;

import lse.neko.robotsim.robot.NekoRobot;
```

```
import lse.neko.NekoProcess;
import lse.neko.NekoProcessInitializer;
import lse.neko.robotsim.algorithm.*;
import lse.neko.SenderInterface;
import lse.neko.abcast.Lamport;

import org.apache.java.util.Configurations;

/**
 * An initializer class for robots
 * @author Smath
 *
 */
public class RobotSimInitializer implements NekoProcessInitializer{

        public void init(NekoProcess process, Configurations config)
                throws Exception
        {
                NekoRobot robot = (NekoRobot)process;
                //algorithm for robot
                PathReservation algo = new PathReservation(robot);
                algo.setId("path_reservation");

                //simulated network
                SenderInterface net = process.getDefaultNetwork();

                //lamport total-ordered broadcast
                Lamport abcast = new Lamport(process, 0);
                final Object abcastId = "abcast";
                abcast.setId(abcastId);

                //protocol composition
                abcast.setSender(net);
                abcast.setReceiver(algo);
                algo.setSender(abcast);

                //launch robots and protocols
```

```
                abcast.launch();
                algo.launch();
        }
}
```

6.3 Available Motion and Sensor Commands

In this section, we will list all available motion and sensor commands those are equipped to robots as a default modules. Please note that, developer can always extend their new own motion and sensor modules. As long as they respect interfaces providing in the framework, those extended modules should work flawlessly. These commands are ready to be used in the algorithm class that extends *lse.neko.robotsim.algorithm.AbstractAlgorithm* abstract class.

6.3.1 Motion Commands

- $actuator.forward(movespeed, distance)$
- $actuator.turn(movespeed, distance)$
- $actuator.moveTo(destination, movespeed, turnspeed)$

6.3.2 Sensor Commands

- **GPS sensor: PULL**
  $gps.getSelfPosition(errorFactor)$: return Point2D.Double
  $gps.getAllPosition(errorFactor)$: return Point2D.Double[]
- **View sensor: PULL**
  $viewSensor.getRobotsInView(viewSize)$: return Point2D.Double[]
- **Proximity sensor: PULL**
  $proximitySensor.getProximityView(viewSize)$: return Point2D.Double[][]
- **Proximity sensor: PUSH**
  NOTE: Interrupt robot(me) if any robot come into viewSize area.
  $proximitySensor.activate(viewSize)$: return Point2D.Double[][]

Chapter 7

Conclusion and Future Works

7.1 Conclusion

With this research, we have developed a simulation engine for cooperative mobile robot on the top of Neko framework as its extension. We have tested many considerable factors in order to provide the most flexibility for developers especially for Neko's developers. We did not just developed a new standalone simulator. We have developed this simulator while seriously considering about how to inherit most of Neko's strong points such as protocol's encapsulation, reusability and many other of its features in to mobile robots world. The works those we have done can be listed as the following.

- Designed and implemented a framework for representing robots and sensors mechanisms in discrete event simulation.
- Solve the problem of how to calculate robots coordinations at time t when we know only t_{start} and t_{finish}. The problem was solved by using parametric equations.
- Developed a collision detector which can correctly pre-calculate collision in the future by using only all robots paths information at current time.
- Solved the problem of how to know the collision time in the future by using parametric equations and robots radius information.
- Designed and implemented sensor system, with the attempt to cover wide range of simulation's situation as much we can. As a result, we came up with pull and push model.
- Defined and solved the problem of conditional events. when a future event is registered, and after that if robots changed their motion paths while this event has not been triggered yet, this event will be canceled and removed. This type of event is called conditional event. Events those can be condition events are arrive event and collision event.

- Developed multiple motion and sensor modules as default modules, so that basic users can use the simulator without having to create any new module.
- Prepared many of interface/abstract class for future-developers, so that they can extend the system without a significant change in the program.
- Designed new pattern of logging file system. Because the traditional logging system in Neko was developed for just only network simulation.
- Implemented a animation module for generating animating visualization from log file.
- Evaluated the simulator by developed many of example applications.
- Ensured and tested the consistency of using Neko's communication on mobile robots application.
- Ensured and tested the using of Neko's microprotocols composition with mobile robots application.

7.2 Future Works

In the last section, I would like to talk about the interesting extendable points for the future development. Some of the extensions mentioned below are in the process of development by the members of our laboratory.

The first thing that should be extended is the implementation of obstacle object.The next one is, the visualization for robots communication. The third is improvement of reusability for motion algorithms those have been written.Same way as the reusability of communication protocol in Neko, we would like to have a feature to allow user to reuse the old motion algorithm cooperating with new algorithm without having to re-write the code.

The last one is, considering more about the usability of interface between mobility and communication. For instance, communication has speed parameter and effect its reachable range.Or communication range may become shorter if the obstacle blocking in the way is very thick.

7.3 References

[1] Peter Urban, Xavier Defago, and Andre Schiper. Neko: A single environment to simulate and prototype distributed algorithms. Journal of Information Science and Engineering, 18(6):981-997, November 2002.

[2] Brian Gerkey, Richard T. Vaughan and Andrew Howard. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems". In Proceedings of the 11th International Conference on Advanced Robotics (ICAR 2003), pages 317-323, Coimbra, Portugal, June 2003.

[3] Cyber Robotics, Webots, fast prototyping and simulation of mobile robots, http://www.cyberbotics.com

[4] USC/Information Sciences Institute, Advances in Network simulation, IEEE Computer, 33 (5), pp. 59-67, May, 2000. Superceeds USC tech report 99-702b

[5] Andras Varga, The OMNeT++ Discrete Event Simulation System, In the Proceedings of the European Simulation Multiconference (ESM'2001). June 6-9, 2001. Prague, Czech Republic.

[6] Xavier Defago, Samia Souissi: Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. Theor. Comput. Sci. 396(1-3): 97-112 (2008)

[7] Rami Yared, Xavier Defago, Matthias Wiesmann, Collision prevention using group communication for asynchronous cooperative mobile robots, AINA'07, pp. 244-249

[8] Samia Souissi, Xavier Defago, Masafumi Yamashita: Gathering Asynchronous Mobile Robots with Inaccurate Compasses. OPODIS 2006: 333-349

[9] Paeter Urban, Sergio Mena, Xavier Defago, Takuya Katayama: Concurrency in Microprotocol Frameworks, Research Report, JAIST, February 2006.

[10] Sergio Mena, Xavier Cuvellier, Christophe Gregoire, Andre Schiper: Appia vs. Cactus: Comparing Protocol Composition Frameworks, Proc. of 22th IEEE Symposium on Reliable Distributed Systems, October 2003.