

AALBORG UNIVERSITY  
ROBOT PROJECT 2008

# Agent Programming - Robot Traffic



GROUP NUMBER  
s502A

GROUP MEMBERS  
Søren Andreas Juul, Kaspar Henrik Moss Lyngsie  
Michael Vandborg, Kim Fiedler Vestergaard  
Torsteinn Sævar Hjartarson, René Bach Gustafson

October 30<sup>th</sup>, 2008

# Contents

<b>1</b>	<b>Problem definition</b>	<b>1</b>
1.1	Problem definition . . . . .	1
<b>2</b>	<b>Analysis</b>	<b>2</b>
2.1	Analysis . . . . .	2
<b>3</b>	<b>Implementation</b>	<b>5</b>
3.1	BDI Agent . . . . .	5
3.2	Frameworks . . . . .	6
3.3	Sweeping module . . . . .	8
3.4	Movement module . . . . .	9
<b>4</b>	<b>Conclusion</b>	<b>11</b>
4.1	Conclusion . . . . .	11
	<b>Appendices</b>	<b>13</b>
<b>A</b>	<b>Code examples</b>	<b>13</b>
A.1	NXT++ example . . . . .	13
A.2	SmartNXT example . . . . .	14

# Chapter 1

## Problem definition

### 1.1 Problem definition

The problem in this mini project is to construct an agent, which is able to do the following:

- 1. Follow a line of a specific color, and take an appropriate action in an intersection area.
- 2. When two or more robots run into the intersection area, the robot should be able to communicate to avoid collision.

It is also required that the robot will be programmed as a BDI agent.

# Chapter 2

## Analysis

### 2.1 Analysis

In order to know the architecture of the construction of our robot, we have to analyze the different scenarios that the robot may experience.

In order for the robot to succeed, the robot will have to be able to do some correct operations for each of the scenarios.

#### 2.1.1 Scenarios

##### Scenario 1:

In this scenario robot1 wants to follow a line with a specific color, it is already at the position of the line, so the line is known by the robot, and therefore the robot needs to follow the line.

Line position found	Yes.
Intersection with another robot	No.
Operation	Follow the line by its color.

##### Scenario 2:

In this scenario robot1 wants to follow a line with a specific color, it is not at the position of the line, and does not know where the line is.

Line position found	No.
Intersection with another robot	No.
Operation	Search for the line by its color.

##### Scenario 3:

In this scenario robot1 wants to follow a line with a specific color, it is already at the position of the line, but it has gotten to an intersection area, where a robot2 also is located at.

Line position found	Yes.
Intersection with another robot	Yes.
Operation	Communicate with robot2, and decide who needs to go first through the intersection area.

### 2.1.2 BDI architecture

By the different scenarios available to our robot, we will be able to construct a BDI architecture for our robot.

The BDI architecture is made to embody the belief-desire-intention paradigm. The idea is to construct a set of beliefs, desires and intentions that we will be able to implement in our robot.

Lets have a look at the following figure:

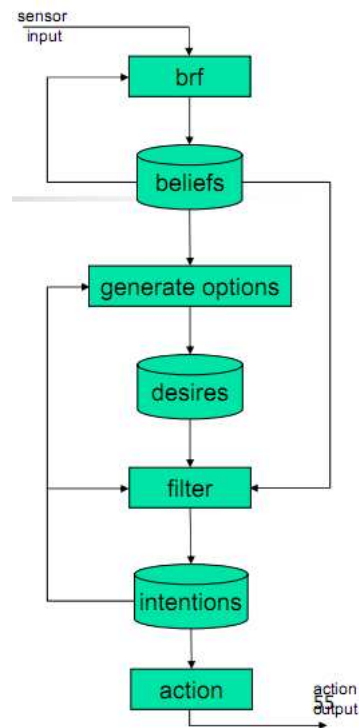


Figure 2.1: BDI architecture

In this model we have some sensor input, from which we will be able to

## 2. Analysis

---

determine a set of new beliefs from all of the beliefs that are available. With these beliefs that our robot currently has, we can generate some options that the robot has which will correspond to its desires. Now a filter function is used, where we determine the intention of the robot, on the basis of its current beliefs, desires and intentions. At the bottom of the figure an action selection function is located, where the robot determine the action to perform on the basis of the current intension.

# Chapter 3

## Implementation

### 3.1 BDI Agent

First of all we have to list some of all the beliefs, desires and intensions that our robot needs in order to work as a BDI agent.

**Beliefs:**

- OnRoad (the robot is located at the correct line)
- IntersectionArea (the robot is located at the intersection area)
- OpponentNear (there is another robot in front of it, some cm away)
- OffRoad (the robot lost the position of the road)
- NeverFoundRoad (the robot has never found the road)

**Desires:**

- Idle
- SearchForRoad
- FollowRoad
- WaitForCommand
- Backtracking

**Intensions:** is a selected desire which the agent has chosen to do.

From these values, we can see that there are three general actions that we need to implement in our robot, based on our desires. First of all we have to

implement an action to follow a line, also we have to implement an action to find a desired line when the position is not known, and last we will have to implement an action to communicate with another robot, in order to get through the intersection area.

#### 3.1.1 How to implement

Based on the beliefs, desires and intensions, we could implement a decision function, which is able to determine what action to execute from the ones we have available by our intensions.

The decision function is shown at the figure below:

**Action:**

1. function *action* ( $p : P$ ):  $A$
2. begin
3.      $B := brf(B, p)$
4.      $D := options(D, I)$
5.      $I := filter(B, D, I)$
6.     return *execute* ( $I$ )
7. end function *action*

Figure 3.1: Decision function

First some input  $P$  (perception) is given, which would be the input from the different sensors on our robot. Then we will generate some beliefs from the perception  $p$ , in section  $B$ . In section  $D$ , we will generate a set of desires and intensions, from the current beliefs that our robot have. Last in section  $I$ , we will generate a single intension from the beliefs, desires and intensions that our robot currently have. This intension then corresponds to a single action that will be executed.

## 3.2 Frameworks

The LEGO NXT block can be implemented in two ways. One is to create programs that can be stored and run on the NXT block itself, making the NXT completely independent. The other way is by passing commands directly to the NXT block through USB or Bluetooth connection. When the program is stored on the NXT block and run, the memory is very limited and the CPU is slower but, the robot is completely independent. By commanding the NXT block directly through a connection gives much more potential

in regards to memory, computing power and usage of databases. But it is completely relying on the connection to hold throughout the execution.

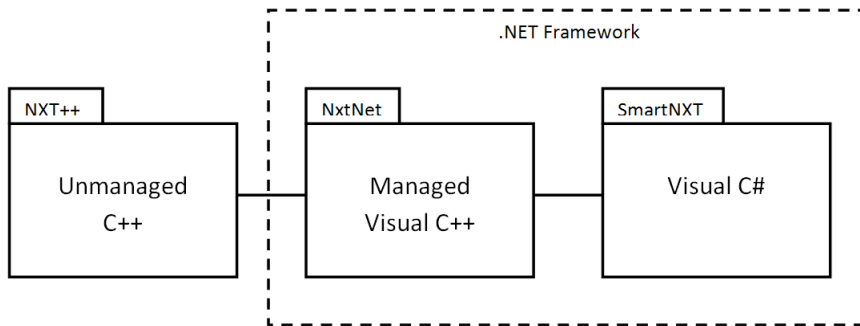


Figure 3.2: The NXT libraries

### 3.2.1 NXT++

NXT++ is an API for the NXT block and surprisingly it is implemented in C++. Though it is good it is redundant code, for instance the connection object has to be passed as a parameter to every command. See code A.1 in Appendix for NXT++ example.

For larger robot applications this gets tedious to write. Complicated algorithms become even more complex and larger than they should be.

### 3.2.2 NxtNet

The NxtNet library was created by the group as a "bridge" from Unmanaged C++ to Managed Visual C++ compatible with the .NET Framework. It only simplifies the method calls and removes some of the redundancy from the NXT++ library. Though it is not fully completed it has enough range to create basic robotics application in .NET.

### 3.2.3 SmartNXT

The SmartNXT library is also created by the group and is written in C#. The purpose of this library is to put the NXT to more object oriented design and make the code more readable.

The `Connect` class is a static class which works as a global class to all the others, thus there is no need to define some connection when other classes are initialized. `Sensor` is the base class for `Touch`, `Light` and `Sonar`. The `Motor` is base for `Arm` which extends the motor for a new purpose, and `Wheels` uses

### 3. Implementation

---

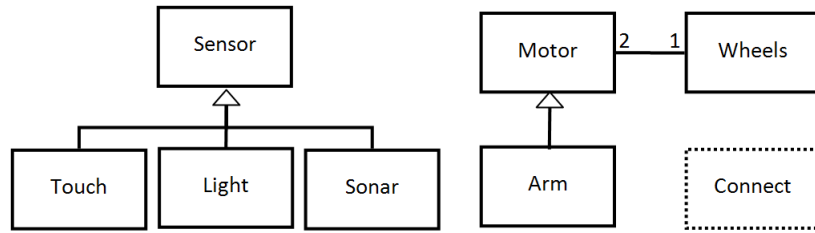


Figure 3.3: The classes of SmartNXT library

two instances of `Motor` to control two wheels, left and right. See code A.2 in Appendix for C# SmartNXT example.

### 3.3 Sweeping module

One of the most important modules of our BDI agent is the sweeping module. This is where the input needed for our robot gets collected.

Mainly we want information about the colors from our light sensor, and the distance to an object from our sonar sensor. This information is to be used by our robot, to know when different beliefs should be obtained and also to know when the information can be used for different actions based on the current belief.

Now lets get into detail about how the module works. The module has five important data attributes that is needed for the light sensor input. These attributes are *startDegree*, *endDegree*, *amount*, *groundColor* and *visited*.

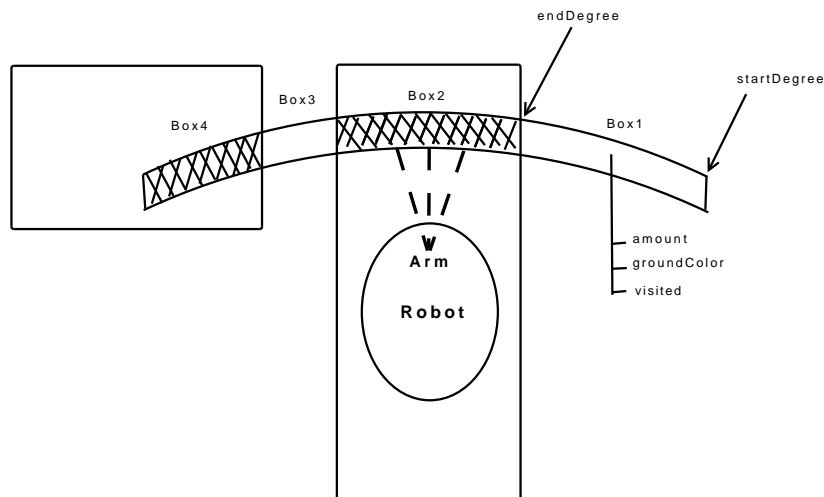


Figure 3.4: Sweeping module

In figure 3.4 we have four boxes, these represents what our robot arm which is moving from the right to the left should be able to locate. Box1 gets represented by a starting degree and an end degree, this tells us the starting position of a specific color and the end position of that color. Each box is therefore a representation of a specific color, which our robot arm has located with the light sensor.

The position of the box is then stored together with its color, amount (degrees) and the visited value. The visited attribute is a Boolean value, which by default is set to false. We then have to set the visited value to true, whenever we are visiting one of the boxes that correspond to the color that we want to follow. This enables us to be able to move to previous not visited boxes, from which we know the position of. So whenever two boxes of the same color which we desire are located, we will visit the box with the greatest amount, and then leave the visited value of the other box to false, so that it may be visited later on.

Another thing that we want is for the sweeping module to maintain information about is the sonar sensor. We need this in order to know whenever an obstacle is in sight, which could be another robot. This is done by having the sweeping module to update the sonar value whenever it is at the center position, and then having some threshold telling the robot when it should react to the object in front of it.

## 3.4 Movement module

We have implemented a movement module, which is responsible for moving the robot in the desired direction, and keeping relevant information about the position of the robot.

As mentioned earlier we were able to implement a sweeping module, which separates the desired colors into boxes that are represented by degrees. Now in order to follow these boxes, we would like our robot to be turning by degrees also.

To turn our robot some amount of degrees, we needed to know how many degrees we had to turn the wheels in order to turn the robot itself. Because our robot has two wheels in the front, which are able to turn independently, we tested how many degrees we had to turn one of the wheels, such that the robot had turned 90 degrees. By doing this test we found that turning one of the wheels 375 degrees, would make the robot turn exactly 90 degrees. Therefore rotating the robot is done by the following equation:  $degree * 375/90$ , which is telling us the number of degrees to turn the wheel.

The move method does only need *degree* as input, which means that every time we move the robot the length that it has moved is fixed. This is done to

### 3. Implementation

---

make the movement much simpler and easier for us to do backtracking. First the method turns the robot if the input *degree* value is different from zero, and then it moves some cm forward. The idea then is that first we do one sweep to find the degrees that we have to turn, then we use the movement module to turn and move the robot, and then once again we sweep and so on.

Because we are able to keep information about previously not visited paths in the sweeping module, backtracking is something that we want possible for our robot. In the movement module we keep information about every move the robot makes, such that we can make the robot move to previous positions. Each movement step is saved with information about *degree*, *rotation* and *moveSpeed*. These steps can then be performed backwards, such that we are able to move  $n$  steps to a previous relevant position.

# Chapter 4

## Conclusion

### 4.1 Conclusion

Throughout this project period, we have experienced a lot of different problems with making an agent which should work in a real world scenario.

When programming something which has to work in a real world environment, reliability becomes a very important factor. Therefore one very challenging thing when implementing our BDI agent was, that the outcome from an action might not always be the same. This could for example be that the color values had changed, or that the movement would be different even though the same amount of power was given to the motors. Another concern has been the scenario itself, where a lot of different problems occurred. This was for example how to handle the communication, as of when the communication should be established and how the robot should be able to tell the server when it is done. Also we thought a lot about how the sweeping would be done, should the arm be moving or should we just move the robot.

In this project there have been different solutions to each problem concerning the robot. Some ideas we had to change because we learned by experience that they wouldnt work. The lesson we learned from this project has therefore been, that working with real world scenarios is not as predictable as we may think, and a lot of decisions has to be made to compensate this.

# Appendices

# Appendix A

## Code examples

### A.1 NXT++ example

```
1:  Comm::NXTComm con;
2:  if(NXT::Open(&con))
3:  {
4:      NXT::KeepAlive(&con);
5:      NXT::Motor::SetForward(&con, OUT_A, 40);
6:      NXT::Motor::SetForward(&con, OUT_B, 40);
7:      NXT::Sensor::SetSonar(&con, IN_1);
8:
9:      // Drive until something is 30cm away
10:     while(NXT::Sensor::GetSonarValue(&con, IN_1) > 30)
11:     {
12:     }
13:
14:
15:     // Take a U-turn
16:     NXT::Motor::Stop(&con, OUT_A, false);
17:     NXT::Motor::Stop(&con, OUT_B, false);
18:     NXT::Motor::SetForward(&con, OUT_A, 40);
19:     NXT::Motor::SetReverse(&con, OUT_B, 40);
20:
21:     MyThread thread;
22:     thread.start();
23:     ::Sleep(500);
24:
25:     NXT::Motor::Stop(&con, OUT_A, false);
26:     NXT::Motor::Stop(&con, OUT_B, false);
27:     NXT::Sensor::SetSonarOff(&con, IN_1);
28:     NXT::Close(&con);
29: }
```

Table A.1: Example robot code in NXT++

## A.2 SmartNXT example

```
1:  if (Connect.USB())
2:  {
3:      Connect.KeepAlive();
4:      Wheels wheels = new Wheels(Port.A, Port.B);
5:      Sonar sonar = new Sonar(Port.Nr4);
6:
7:      wheels.Forward();
8:      sonar.TurnON();
9:
10:     // Drive until something is 30cm away
11:     while (sonar.GetDistance > 30) ;
12:
13:     // Take a U-turn and stop
14:     wheels.Stop();
15:     wheels.Left.Forward();
16:     wheels.Right.Backward();
17:     Thread.Sleep(500);
18:
19:     wheels.Stop();
20:     sonar.TurnOFF();
21: }
22: Connect.Disconnect();
```

Table A.2: Example robot code in SmartNXT