COMP3902 - Robocup

Adam Brimo
abrimo@cse.unsw.edu.au

School of Computer Science & Engineering
University of New South Wales
Sydney 2052, Australia

August 2008

Supervisor
A. Prof. Maurice Pagnucco
K17-412D / +61 2 9385 6925

Assessor
Prof. Claude Sammut
K17-401J / +61 2 9385 6932
Contents

1 Overview 2
  1.1 Background ................................................. 3
  1.2 Document Overview ......................................... 4

2 Behavior 5
  2.1 First Attempt — Python ................................. 7
  2.2 Second Attempt — Ruby ................................. 9
    2.2.1 Reasoning ............................................. 11
    2.2.2 Implementation ................................. 13
    2.2.3 Results ............................................. 16
  2.3 Third Attempt — C++ ................................. 17
    2.3.1 Reasoning ............................................. 19
    2.3.2 Implementation ................................. 20
    2.3.3 Results ............................................. 27
  2.4 Conclusions on Behavioral Structure ......................... 29

3 Competition in Suzhou 30
  3.1 Setbacks .................................................. 31
  3.2 Lessons Learned ........................................ 32

Bibliography and references 33
# List of Figures

2.1 The soccer field for the Standard Platform League. All units are in millimeters. .......................... 6

2.2 A diagram provided by Aldebaran showing the safest method of accessing a module when running the code as a broker. .......................... 9

2.3 A diagram showing the way in which the code is executed on the Nao. . . . 11

2.4 A diagram showing the requires / extends structure of the Ruby behavior code. .......................... 13

2.5 A diagram showing the separation of the different parts of the system. . . . 17

2.6 A diagram showing the way in which the code is executed on the Nao. . . . 19

2.7 A diagram showing the requires / extends structure of the C++ behavior code. .......................... 21

2.8 A state machine diagram showing the process of tracking the ball. . . . . . 22

2.9 An example of a frame from the Nao’s camera showing the location of the ball. .......................... 23

2.10 An example of the angle between the x-axis in the frame of view and the ball. 23

2.11 A diagram showing the robot side-steping around the ball. . . . . . . . . . . . . 24

2.12 An image showing the possible path the Nao could take to reach the ball. . . 25

2.13 A use case diagram showing the testing cycle. .......................... 29

3.1 A photo of the Aldebaran Nao at the competition in Suzhou. .......................... 30
Chapter 1

Overview

The development of the behavior code for the 2008 Robocup competition went through three major revisions and redesigns. This was due in a large part to the extended delay in receiving the Nao robots and the uncertainty over the API for interacting with those robots.

There were three major revisions to the behavior code and accompanying each revision was a change to the architecture of the system. In the original process of porting the previous competitions’ code for Aibo robots to the new Nao robots it was determined that the behavior code would have to be rewritten because the new robot’s behavior would be very different. However the path of using Python for the behavior code and C++ for the general code-base was continued until the API for the Nao robots was changed.

The simulator had provided a C++ API for interacting with the joints of the robot directly while an interface between the C++ code and Python code had been developed in previous years and could be generated with some open source tools. When the first version of the Nao arrived, the C++ API was removed and replaced by a layer called NaoQi. This layer managed all of the joint angles and movements of the robot however connecting to this module was done over TCP, although the module could be accessed from either C++, Ruby or Urbi. As explained in this report, the behavior code was then moved to Ruby and due to the constraints of NaoQi the architecture was also changed from a cycle based approach to an action based approach. However after testing and development of the Ruby behavior code it became apparent that behaviors had to run faster in conjunction with the whole code-base. Therefore behavior code was moved to C++ and changed back to being executed every cycle. This is explained further in the section on the C++ implementation of behavior.

A number of challenges were encountered involving the testing of code and the integration of different modules. Due to a number of hardware issues also mentioned in this report it was not possible to properly or thoroughly test all of the behavior code at each stage of development.
1.1 Background

The 2008 competition marked the first time that humanoid robots were used for the standard platform league. This presented a number of challenges, including the porting of the previous vision code and redeveloping locomotion, behavior and the sensor information module to work with the new robots. The robots themselves were expected to arrive in the beginning of 2008 however the date was pushed back continuously by Aldebaran Robotics until the first beta version of the robot finally arrived in the middle of May.

The first version of the Nao that arrived had a number of issues including faulty plastic washers and soft plastics gears that easily wore away with moderate usage. After having the robot a few days and testing some sample code, the robot fell over causing the head to fall off as it had not been secured properly. A few days later one of the legs broke off after a crash in the supplied software caused the joints to be set to incorrect angles.

Working and usable versions of the robots did not arrive until mid June 2008, less than two months before the competition and over four months later than projected. However of the four robots the university had ordered, only two worked out of the box while the other two remained broken and could not be fixed. These issues hampered the development of every module, especially that of behavior and locomotion. Without working robots the locomotion code could not be developed and tested while the behavior code could not be tested thoroughly until the vision and locomotion modules were in a working order.

Throughout this process there was a simulator available for use, Webots, developed by Cyberbotics, was released to all of the teams in advance. The main problem with using the simulator was that the interface for communicating with the robot was completely different to the actual API provided with the first robot. Therefore it would be nearly impossible to develop locomotion or behavior code in the simulator that could be translated to work directly on the Nao.

This set of issues and challenges led to a number of changes to the system architecture and the way behavior was developed and structured. The lack of a reliable robot to test the behavior code led to issues with unexpected behavior and results across all the modules.
1.2 Document Overview

This document will cover the different stages and implementations of the behavior code throughout the rUNSWift 2008 season. The testing and development of the behavior code was largely affected by the integrity and durability of the robot platform and as such some of the hardware issues encountered are also discussed. The following is a short summary of what is included in each of the major sections.

**Behavior** The overall structure of the behavior code and the changes in implementation language and system architecture are discussed in detail.

- **First Attempt - Python** The initial development of the behavior code using the Python scripting language.
- **Second Attempt - Ruby** The change in the API for the Nao robots brought with it some required changed to system architecture and structure.
- **Third Attempt - C++** Development of behavior was moved to C++ to improve the speed and flexibility of behavior.

**Competition in Suzhou** The competition presented a number of challenges involving teamwork and development as well as hardware issues.

- **Setbacks** A number of issues arose during the competition that hindered the development and testing of the code-base.
- **Lessons Learned** Based on the issues and setbacks that occurred during the competition as well as problems that had occurred throughout the year, a number of lessons were learned.
Chapter 2

Behavior

The development of the 2008 behavior code in the UNSW Robocup code-base went through three major revisions in which the development language and code structure changed. The decision to move from one development structure and language to another was due to the changing interface between the rUNSWift code-base and the Nao robot as well as gains in performance and flexibility. The three main revisions listed below are described in this chapter.

**First Attempt - Python**  The original Aibo code-base used the Python language for all behavior code hence it was the starting point for the new behavior code for the Nao robots.

**Second Attempt - Ruby**  The API for the Nao robots was supplied in three languages, Urbi, C++ and Ruby. The native support for Ruby by the Nao made it the most logical choice.

**Third Attempt - C++**  Performance and changing interfaces within the rUNSWift code-base were the main reasons that development of the behavior code was moved to C++.
Figure 2.1: The soccer field for the Standard Platform League. All units are in millimeters.
2.1 First Attempt — Python

The majority of the code-base for the Nao was ported directly from the Aibo code that had been developed over the course of the past ten years. The python behavior code was much more recent, having been developed in 2004 to replace the previous C++ implementation of behavior. Ideally the scripting of behavior in Python would allow more rapid changes to be made and commands to be sent over the network to control the robot.

In order for the Python behavior code to be integrated into the C++ code-base an interface layer would have to be developed that translated the C++ types to Python types and allowed the methods to be called from each language. This process is called embedding and extending - as the behavior code would call the vision, localization and locomotion methods while the main program in C++ would call a Python method to execute the behavior each cycle.

An interface between the C++ methods and the Python code had to be developed, in the Aibo code-base this was done manually for each of the methods that were required. However the code-base at the time was sufficiently developed and polished that few changes to the interfaces would be required. Furthermore the lack of robots or an API for the robots resulted in constant changes to the interfaces of vision, localization and locomotion. It was then logical to presume that the interfaces may continually change up until the competition and any manual means of developing the interface between the two languages could result in bugs that would be time consuming to resolve.

Interface Generation

At this stage an investigation began into the automated means of generating an interface between Python and C++ based entirely on the header files of the C++ classes. The most common and full featured interface generator between scripting and programming languages is arguably SWIG (Simplified Wrapper and Interface Generator).

“SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code. In addition, SWIG provides a variety of customization features that let you tailor the wrapping process to suit your application.”

During initial testing with C++ and Python sample code, SWIG was able to generate an interface which allowed the C++ functions to be accessed in Python. Some code was also developed to call the Python methods from C++, both the extending and embedding

---

of the code appeared to be working fine in rudimentary examples. However once the principle was applied to the larger code-base, namely vision and locomotion, SWIG had some difficulty converting custom C++ types into Python objects.

Another issue encountered with SWIG was that it was unable to properly deal with custom namespaces or use those namespaces to generate the Python interfaces. Initially it was believed that this was an issue with sending commands to the robot (through the simulator) so another test scenario was constructed. This test involved generating an interface for the Nao API as supplied with the Webots simulator. In this test the entire interface for controlling the movement of joints as well as the text to speech methods for the Nao were wrapped for use with Python. This test proved successful however the interfaces for the API did not make use of any namespaces, inheritance or complex types.

Conclusions

Due to the complexity of the code it did not seem possible to efficiently or easily generate the required interfaces for each of the modules in the code-base. More time would have to be spent on ensuring the interface between the two languages worked properly, which would take time away from the actual development and testing of the code.
2.2 Second Attempt — Ruby

The second attempt at behavior was implemented in the Ruby scripting language. However a major change was also made at this stage of development. Previously, the behavior code was integrated into the general code-base and run in its own thread. However the new API for the Nao provided a new method of interacting with the robot; this new method allowed the rUNSWift code to be accessed in Ruby without the need for a translation layer between Ruby and C++. This is done by specifying an interface in C++ and then registering that interface with NaoQi, hence NaoQi creates a module that includes those specified methods.

The NaoQi software provided by Aldebaran is the only supported means of controlling the movement and reading information from the Nao. The functionality of the system was divided into modules supplied by Aldebaran however the facility was provided to create additional modules as required. The following significant modules were provided with NaoQi, the corresponding descriptions are according to Aldebaran.  

![Diagram](image)

Figure 2.2: A diagram provided by Aldebaran showing the safest method of accessing a module when running the code as a broker.

**ALMotion** "The motion module was designed to facilitate the control of Nao, going way beyond simple joint-space commands to allow direct control of end effectors, directly manipulate the center of mass and request high level motions such as ‘walk straight 10cm’.”

---

2Based on the Aldebaran online documentation for the Nao Ware Documentation for the NaoQi SDK.
**ALVision** “The VideoInput module is architectured in order to give every module related to vision an access to the camera buffer through a specific provider named Layer Extractor Module (L.E.M.). This L.E.M. locally changes the original camera stream into a given stream and transmits it to the vision module which can be local or remote. Each vision module has its dedicated L.E.M.”

**ALTTextToSpeech** “The TextToSpeech module allows the robot to speak. It sends commands to a text-to-speech engine, and allows voice customization.”

**ALLeds** Provides direct control of the LED lights on the Nao. The LEDs are organized into groups so that several LEDs can be controlled with a single command.

A module for the rUNSWift code-base was created that provided control of the robot and also allowed data to be exchanged, including vision and localization data. This module could then be accessed from the Ruby behavior code by connecting to the module through NaoQi.

This change to the way the behavior interacted with the rest of the code base also changed the structure of the behavior code from a cycle based approach to an action based approach. At this stage a large amount of strategy was developed for game play as well as approaching and finding the ball.
2.2.1 Reasoning

Moving from Python to Ruby

The decision to use Ruby as the scripting language was due to the change in the API provided by Aldebaran. During the initial stages of development - the Webots Simulator was used in conjunction with a C++ API for accessing and sending information to the robot. However the arrival of the actual robots and the removal of the C++ API forced a change to the scripting language being used.

The ability to create a module out of the existing robocup code being developed provided a simple way of accessing the required methods. This eliminated the need for an interfacing layer to be written between the scripting language and language of the codebase. However this method of creating a module was only available to C++, Ruby and Urbi.

It would still be possible to use Python although this would require an internally developed interfacing layer for C++ functions that would be continuously changing due to time constraints. Therefore, under the given time constraints - the only plausible choice for a scripting language was Ruby.

Action Based Behavior

In the previous architecture, the behavior code was run each cycle along with every other thread in the program. However this required that the program be run as a C++ binary, either as a broker or a dynamic library, rather than the entire program being started and run through Ruby. Figure 2.3 shows this new method of executing the program. Essentially the Ruby behavior code calls the rUNSWift module that exists within NaoQi, which then requests the information from the C++ layer and sends it back through NaoQi to the behavior code.

Figure 2.3: A diagram showing the way in which the code is executed on the Nao.
Action based behavior also had the advantage of inherently facilitating decision making for the Nao. Rather than running all of the behavior code at each cycle in the program, only the areas of the vision and localization would be queried when they are required. However, the ability to interrupt the action or decision still exists in this approach - therefore the robot would not continue walking to a determined location if the ball had moved across the field.
2.2. SECOND ATTEMPT — RUBY
CHAPTER 2. BEHAVIOR

2.2.2 Implementation

System Architecture

A hierarchical structure was developed for the behavior code with the top level being a single file which starts and runs the program, this is shown in Figure 2.4. The game could be in any of six different states according to the rules of the game. Therefore the StateManager class was developed to determine which state the game was in; for example, a state could be ‘Playing’ where the robot would be able to move freely and play a game whereas in the ‘Ready’ state the robot would not be allowed to move at all.

![Diagram of System Architecture](image)

Figure 2.4: A diagram showing the requires / extends structure of the Ruby behavior code.

The StateManager linked to the GamePlay class which would determine what actions should be done at a given point in time. This included the making of decisions on which actions should be taken, the following are examples of decisions:

**Should I go to the ball?**  Would determine if the robot should go to the ball.

**Should I try and intercept the ball?**  Determines if the robot should move from it’s current position towards the opposing team’s robot.

**Does our team currently have the ball?**  Determines whether the team has posses-
sion of the ball or is closest to the ball.

**Should I defend the goal?** Determines if the robot should move into the goalie box and defend the goal.

**Should I kick the ball?** Decides if the robot should kick the ball in the current direction.

The central part of this system architecture is the Player, which abstractly represents the robot. The abstract type Player provides all of the basic functionality for playing the game however it can be extended by either a Goalie or a Striker. By utilizing different types for each of the different roles, the decisions made by the robot can be different depending on the role of the robot. The goalie would also provide specific skills which would only be applicable to that role. For example, the goalie would generally never leave the goalie box so it would make sense for the goalie to always side step rather than walk to the ball at some point on the field. While the striker would generally want to go directly to the ball and chase the ball around the field.

Each player would hold all the knowledge relevant to it’s location on the field as well as the location of the ball. This information would come directly from the rUNSWift module that was sending data through NaoQi. By allowing the player to store this data, it becomes far easier to query the location of the robot for a specific player. Additionally, the player would also contain an instance of the movement class that would interface directly with the rUNSWift module to control the movement of the robot.

**Player Skills**

The actions a robot could take were named skills and each player would have a generic set of skills that would be used for playing a game. The majority of these skills were based heavily on localization and required the robot to have an accurate knowledge of the location of the ball as well as it’s own location. The following segment of code shows how the robot would move towards a ball using the methods available in movement and location.

```ruby
# if the robot knows the location of the ball then move
# towards that location.
def goToBallAtPoint
  @moveInstance.turn(
    @locationInstance.calculateTurn(
      @locationInstance.locationOfBall))
  @moveInstance.move(
    @locationInstance.getBallDistance, 0.0)
end
```

Adam M. Brimo 14 October 9, 2008
However the localization data provided is generally not accurate and in many cases no data is provided for the location of the ball or the location of the robot. In this instance the robot would need a method of moving towards what it thinks is the ball. This would also be used if the readings from localization have a very low confidence rating or fail some general bounds checking.

```ruby
# keep looking for the ball and walking towards it
def findAndWalkToBall
    @locationInstance.myLocation
    ballHeadYaw = lookAtBall
    pointToBall(ballHeadYaw)
    @moveInstance.waitUntilFinished
    ballDistance = @locationInstance.getBallDistance
    if ballDistance != 0.0
        @moveInstance.move(
            @locationInstance.getBallDistance, 0.0)
    end
    while @locationInstance.ballInFrame &&
        @moveInstance.isWalkActive
    end
    @moveInstance.stopWalking
end
```

This method of finding the ball involved the robot looking around and scanning until it could accurately determine that the ball was in sight. At this point the head yaw of the robot would be recorded and the robot would turn it’s entire body so that it would be facing directly at the ball. The distance to the ball was then calculated in the rUNSWift module through the vision code and passed back to the Ruby behavior. Once this was completed the robot would walk towards the ball until it was no longer in sight.
2.2.3 Results

The development and implementation of action based behavior using Ruby had some unexpected results. The main issue arising from the use of Ruby or any scripting language in general was the processing power required to run the interpreter. When running the behavior code by itself, it would generally use between 20-30% of the CPU with NaoQi taking up all remaining processing power. However this result was achieved without running the vision module with the behavior code on the robot. Once the entire rUNSWi ft library was loaded on the robot and the whole system was run together, vision would achieve under five frames per second and the robot would have difficulty performing any actions. This was due to the requirement that NaoQi have a significant portion of the memory and processing power in order for locomotion to run effectively.

The speed at which the system would run was unacceptable and showed that either the behavior code was performing too many calculations or that another implementation must be tested. In order to determine if the problem was with the behavior code or the running of the interpreter, NaoQi and the rUNSWi ft code concurrently; a test was carried out with the absolute minimum Ruby code. This test involved sending a single command to move one of the joints on the robot to a specific point and then back to it’s original position. The CPU usage was still around 20% and the movement of the arm was not smooth.

The move from cycle based behavior to action based behavior did not provide promising results. Although the system could not be run as a whole and all of the testing was based around auto-generated vision data, it became clear that basing the behavior on specific actions rather than updating every cycle yielded inconsistent results. In this implementation of behavior, all vision and localization data had to be requested during each action. This meant that it was not possible to check the if the goal was in sight at every cycle because there was no concept of cycles.
2.3 Third Attempt — C++

Based on the results of the behavior code developed in Ruby it became clear that the architecture of the behavior code and the code-base in general needed to be modified. Using an action based approach for behavior caused problems with correctly managing and switching between states and also the ability to change a decision once it had been made. Additionally the use of a scripting language proved to be very slow on the robots and caused the entire system to run very slowly. This led to the realization that behavior would have to be implemented in the same language as the rest of the code-base.

The code-base in general was developed in C++ and all of the interfaces between vision, localization and movement already existed. Therefore there would be no need to develop a module through the NaoQi architecture and send all of the data through NaoQi, which would send it over TCP.

As the behavior code would be integrated into the general code-base it would have to be run in it’s own thread. This also meant that the behavior would be done in a cycle based approach as was originally planned and done in Python. However the major difference between the previous implementation in Python and the new implementation in C++ is the behavior is run in the same thread as locomotion.

![Diagram](image)

Figure 2.5: A diagram showing the separation of the different parts of the system.

The diagram in Figure 2.5 shows the different areas of the system. Each area was developed as a separate module with vision and localization sharing data with the behavior layer. As mentioned previously though, the locomotion module would only be called by the behavior layer. This is due in a large part to the way that NaoQi handled the motion commands going to the robot. The movement of joint angles and the general walking of the robot are implemented within the NaoQi layer that runs in it’s own separate process.
Therefore in this system architecture there are three threads that are part of the rUNSWift code-base, including vision, localization and behavior / locomotion. Additionally there is the NaoQi process that runs in the background, which manages all of the interaction between the robot and the rUNSWift program.
2.3.1 Reasoning

There are a number of main reasons that behavior was moved from Ruby (or any scripting language for that matter) to C++. This includes:

- The interpreter for the scripting language requires a considerable portion of the processing power available to the robot. With the processing constraints on the robot, even the most simple Ruby or Python program would require at least 10% of the CPU.

- NaoQi requires a large amount of the processing power of the CPU and will generally try to use as much as possible.

- The locomotion and joint angle movements of the robot will not work properly or smoothly when there is very little available processing power.

- Including a number of improvements and optimizations, the vision processing on the Nao takes considerably more power and time than on the Aibo robots. This is because the camera on the Nao captures higher resolution images and that more scan lines have to be applied to each image because of the increased height of the robot and the height of the goals.

After developing the action based behavior code it became clear that the behavior code should re-evaluate the decisions being made immediately as new information becomes available. This also greatly simplified the system architecture to the point where a minimal number of calls to the NaoQi interface were required.

![Diagram showing code execution on Nao](image)

Figure 2.6: A diagram showing the way in which the code is executed on the Nao.

Figure 2.6 illustrates how the call structure between NaoQi and the rUNSWift code-base has been simplified. This resulted in a much faster execution of the program and a noticeable speed-up in the number of cycles per second.
2.3.2 Implementation

The structure of the behavior code was developed around the concept of an actual soccer player. In the code, the uppermost layer was the player, which is directly called from the behavior thread. The decision making for the player was controlled by the player’s brain, however the brain was an abstract class that was extended by any one of ‘Striker’, ‘Defender’ or ‘Goalie’. The main assumption that was made in this code structure was that the role of the robot would not be changed during a game. Therefore dynamic role allocation was not possible with this architecture, although it was not necessary in the 2008 competition.

Default Actions of the Robot / Tracking

Each cycle the robot would receive new information about it’s surroundings, what it can see and the current position of all of it’s joints. Therefore at any given point in time the robot should be performing an action, this was called the default behavior state of the robot. However the behavior in this state was determined by the role of the robot, this is because a goalie would behave differently if it saw the ball than if the striker saw the ball.

In this default state, (for a striker) the robot would be continuously searching for the ball. This process, entitled tracking, was broken down into three main states that included: seeing the ball, having recently seen the ball and not seeing the ball for a given number of cycles. The state machine diagram in Figure 2.8 shows the steps in the process of tracking the ball.

The process shown was executed every cycle that behavior was run, therefore it was highly likely that the state could change rapidly. In order to deal with this issue and additional issues involving false positives relating to seeing the ball, the intermediate state of having recently seen the ball provided a buffer zone for the decision making. In this state, the robot would move it’s head in the direction it thought it had seen the ball last. Therefore if the ball was moving out of the robot’s frame of view and the last known location was in the top left hand corner of it’s vision; the robot would move it’s head upwards and to the left in order to try and locate the ball again.

Once the robot had successfully seen the ball at some stage in the frame it had to ensure that the ball was in the center of the image frame. However after some testing it was determined that the robot would never be able to perfectly center the ball in it’s frame of view. This is due to the inaccuracy in the calculations of the center of the ball when it is moving or even when it is stationary as background noise may affect the recognition of the ball.

To deal with such issues, an acceptable margin of error was implemented for the center of the frame of view. The robot would then only try and adjust it’s head to center the ball if the ball moved outside a margin of error in the middle of the frame. This margin
Figure 2.7: A diagram showing the requires / extends structure of the C++ behavior code.
Figure 2.8: A state machine diagram showing the process of tracking the ball.
was initially set to approximately 5% of the resolution of the image however after further testing on the level of noise and the ability for vision to correctly determine the center of the ball, that margin was changed to approximately 20% of the resolution in the center of the frame.

Figure 2.9: An example of a frame from the Nao’s camera showing the location of the ball.

At each cycle the behavior code was run, slight changes would be made to the movement of the head yaw and pitch. As shown in Figure 2.8, the state machine diagram, once the ball is clearly visible, the behavior code calls the method: focusOnBall(). This method would perform the minute adjustments to the position of the head to try and move the head based on the center of the ball. Centering the ball within the margin was accomplished by determining the angle of the offset from the ball to the centre point in the frame. An example of this is shown in figure 2.10.

Figure 2.10: An example of the angle between the x-axis in the frame of view and the ball.
Robot Skills

The diagram shown in Figure 2.7 shows the relationship between each of the classes in the behavior code. The brain contained a method entitled ‘think’ that would determine the next action of the robot using the general set of ‘Skills’ available.

Getting Behind the Ball In order for the robot to accurately align itself in the correct direction facing the goal, the robot would have to move around the ball. However it was not possible for the robot to arc side-step around an object using the walking motions supplied by Aldebaran or developed in the general code-base.

![Diagram showing robot side-stepping around the ball.](image)

Figure 2.11: A diagram showing the robot side-stepping around the ball.

Therefore in order to accomplish this task, the robot would side-step a minimal distance and then turn a calculated number of degrees to re-align with the ball. This took advantage of the default behavior developed for the player to create this arc side-step. Using this behavior, the robot automatically adjusted it’s body to align with the ball if the angle of the ball relative to the body was beyond a certain threshold. Therefore, when the robot had side-stepped a sufficient distance, it would stop, turn, then continue with the behavior until it had seen the goal or another condition for attacking was met. This behavior of walking around the ball is shown in Figure 2.11.

Line up with the Ball In order to simplify the process of lining up to the ball, this skill assumed that the robot would already be in front of the ball with less than 50cm of space between the ball and the robot. Hence lining up with the ball would be done after the robot is behind the ball and is facing the goal or has otherwise stopped in front of the ball.

The robot would line up with the ball by first trying to move as close to the ball as possible. If the ball was no longer in sight then the robot stopped walking and then bent down to determine how much further away it was from the ball. At this point
the robot would also try and calculate how far to the left or right of the ball it was and side-step that small distance so that it was properly aligned for kicking.

This method of lining up to the ball was never completely tested or used as it relied on the robot being able to bend down and look at the ground. That ‘bending down’ routine was never developed sufficiently for use on the actually robot and therefore there was no working method of properly lining up for the ball.

**Walk to a Point on the Field** There were a number of ways for the robot to get from one end of the field to another. The implementation of the behavior code used a system of weightings / costs to determine which method of getting to a given point on the field would be the most efficient.

For example, for the Nao to move from one side of the field to another, provided it is facing in the opposite direction to the point it wants to move to, it would have had to calculate the best route. This was done in a fairly simple manner in this initial implementation because the only methods of movement that were available were an arc walk, walking straight and turning. Thus to get to that point on the field the robot may be able to turn then arc walk or walk then turn then walk again or any number of combinations.

![Diagram showing possible path](image)

Figure 2.12: An image showing the possible path the Nao could take to reach the ball.

**Walk a Given Distance** This skill allowed the robot to walk a given distance in conjunction with a possible turn. Therefore the robot would be able to either walk in the direction it is pointed or walk in a given direction by first turning and then walking straight. However this method also utilized a similar formula to the method
which allowed the robot to walk to a point on the field. Essentially this method allowed the best path to be calculated by providing only a distance and a heading that represented the direction in which the robot would have moved.
2.3.3 Results

The third attempt at developing the behavior code solved many of the problems experienced in the Python and Ruby implementations. As no interpreter for a scripting language was required, the code was actually able to run very quickly and the cycle time was approximately twenty milliseconds. However the behavior code was run in sync with the cycle time of vision because it would be unnecessary to rerun behavior if the information is exactly the same as the previous cycle.

API for Modules

Due to the behavior code being written in the same language as the rest of the modules, it was very easy to access all of the methods in each module whenever required. This was very convenient however the vision and localization code was continuously being updated and in some cases the API was changing as well. At one stage the coordinate systems that were being used for reporting the coordinates of the ball in the frame of view were shifted around. This caused large problems for the behavior code so an interface between behavior and vision / localization was developed to ensure that those modules are only referenced in a single class in the behavior code.

The interface allowed for modifications or calculations to also be done on the data being received from vision before it was used in the rest of the behavior code. This also served as a method of caching all of the vision data for each cycle ensuring that the same calculations wouldn’t be run multiple times unnecessarily. Having this separate interface also proved beneficial when the public methods of the area of vision dealing with the ball’s location on the frame were renamed and the calculations where modified. By using the behavior interface for vision to modify those calculations as they arrived it was possible to ensure that the behavior code was receiving the data that was required.

Skills that Worked Well

Ball Tracking The ball tracking developed for behavior worked very well; once the robot had seen a tiny part of the ball it was able to determine which direction to look and bring it into focus within a couple of cycles (provided the head pitch and yaw were able to move the given distance in a cycle). It was originally noticed that the behavior would be delayed by one cycle because the joint angles being received from NaoQi would not be up to date with the vision data being received and sent back to behavior. However this was solved by caching the joint angles of the robot through the vision module and then accessing those joint angles instead of the ones coming directly from NaoQi. This ensured that the actions of the behavior would be in sync with vision- thus allowing the robot to track onto the ball very quickly.
**Getting Behind the Ball** The ability for the robot to move around the ball worked particularly well despite the inaccuracies of some of the movements of the robot. The Nao was able to move around the ball in a shape resembling a polygon while keeping between forty and fifty centimeters from the ball at all times.

**As a Whole** The entire behavior system did actually work although there were a number of issues with approaching the ball after the robot had properly lined up. The cases for approaching the ball generally included seeing the goal at some point in time and that was used as the basis for attacking. However the ability to recognize the goal was not entirely implemented and tested due to time constraints and therefore the robot had trouble making the final approach to the ball. To deal with this issue a number of special cases were developed such that the robot would decide to approach the ball even if it had not seen the goal, under the assumption that it was facing in the correct direction. However it was evident that more of these cases were required to ensure the robot would actually approach the ball.
2.4 Conclusions on Behavioral Structure

In the testing of both action based and cycle based behavior it was apparent that the behavior code would have to be run every cycle at the same speed as vision. Mainly this is the case because the behavior module should only be re-evaluating a decision as new information is being received. In the previous Aibo code base this would have caused a problem because the behavior had to run independently of the locomotion. However due to the structure of NaoQi - all motion commands are automatically run in a separate process and independent of the general code-base.

The development of behavior in C++ provided more benefits than any other solution. Having the code run within the rUNSWift library ensured that there would not be any slow-down due to the use of an interpreter. It also provided the ability to more easily transfer data between the vision and localization modules as there were no restrictions on the types which would be passed around. This contrasts greatly with the Python and Ruby implementations, as in Python it would be required to convert each type from C++ to Python which takes up more time and processing power. Using NaoQi and running the rUNSWift code-base as a module provided even more restrictions on types and only basic doubles and strings could be used in most cases.

![Testing Cycle Diagram](image_url)

Figure 2.13: A use case diagram showing the testing cycle.

The only difficulty in developing behavior in C++ is that testing the smallest changes to the code requires that the program be recompiled, transfered to the robot and then the robot would have to be rebooted. With multiple people working on testing and development of the code - the time taken to compile, transfer and reboot NaoQi was not that significant although at the competition the lack of working robots reduced the overall amount of time available for testing.
Chapter 3

Competition in Suzhou

The competition in Suzhou was one of the first times the entire team was together in the same place for an extended period of time. Each group of people worked on their own module and tested each module in isolation or by walking through the code. The first two days were marred by the lack of any robots to test behavior or any modules in the code-base. This led to substantial problems and bugs as a large amount of development was done over the week of the competition and very little of it could be tested, even in isolation. Therefore it became difficult to properly test behavior when the underlying modules were not thoroughly tested.
3.1 Setbacks

The week long competition was marked by a number of setbacks in both the development and testing stages. This also included the improper use of subversion and the inability to properly test the code being developed.

Broken Robots

The team arrived at the competition without any working robots and initial estimates of repair times were inaccurate as Aldebaran has underestimated the number of faulty robots and the time it would take to repair each of them. For the first two days at the competition no testing could be carried out on any areas of the code and when one of the robots did return from the repair center, it broke again shortly afterwards.

After starting with four robots, the team never had more than one robot fully functional and at most times there were no functioning robots available. This led to a large amount of untested code being run for the first time in a game thus leading to unexpected results. Working robots were generally only supplied for the games and once those were completed the working robots were returned to Aldebaran.

SVN & Untested Code

A major issue that led to a significant amount of lost time and set back development was the committing of code to subversion that did not compile or did not run. Due to the large amount of development being done at the competition, each person needed to have the most up to date version of the code in order to develop their own modules. However the amount of the development being undertaken was disproportional to the amount of testing being done. Significant changes to the code such as changing the calculations of offsets and angles from being relative to the robot’s body to being relative to it’s head were committed to subversion without being tested with the system as a whole.

Significant changes to the underlying modules that controlled vision and localization had dramatic effects on the behavior code. At one point some of the updates made to vision and the calculation of angles were incorrect and provided largely incorrect data back to the robot regarding the location of the ball in the robot’s frame of view. However this was not discovered until after two days of reviewing and rewriting the behavior code to determine that the problem did not lay in the behavior code but rather in the vision module.
3.2 Lessons Learned

Develop Coding Guidelines

This year involved the porting of a significant amount of code from the Aibo robots to the Nao as well as the development of new modules such as locomotion and behavior. This led to a variety of different naming conventions being used for both files and functions depending on the individual who originally coded the file or the latest person to modify it. In the midst of a competition, when trying to find the right method or object to use, it can be very inefficient to continually lookup what naming convention is being used. Therefore in the future there should be a standard way of naming files, classes, methods and variables.

Maintain a Stable Subversion Repository

In order to rectify one of the major issues encountered during the development of the robocup code, a policy regarding committing code to subversion needs to be adopted. In many cases a large number of minor changes are committed to subversion without proper testing as some people merely assume that the code works. However code that compiles is different to code that runs; this distinction was not made in the development of the current code-base. It was especially obvious during the competition when hundreds of commits occurred in the span of a few days and multiple bugs were introduced at different stages in that process.

Testing and Development Cycle

The development of the code was seperated based on the modules each person was working on. Therefore the group of people working on locomotion would only be editing the code for that module. Although this seemed logical and it would still be the case that those people working on a given module only modify the code in that module, it is important that the module is tested with the entire code-base. This is something that was lacking from the development cycle this past competition. There existed an assumption that so long as the module works in isolation then it will work in all cases however this was not the case as was discovered during the competition. In the future it would be important that the system is tested as a whole even if the changes made are only part of a certain module.
Bibliography


