THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE &
ENGINEERING

COMP3902 Special Project B
Report

Author:
Kim Cuong Pham, 3044938
kimpham@cse.unsw.edu.au

Supervisor:
Dr. William Uther
willu@cse.unw.edu.au

Submission date: 22nd September 2004

Table of content

1. Introduction

RoboCup project is a world-wide research initiative in the field of AI and robotics. At UNSW, RoboCup is an on-going project that has undergone in the last 5 years. The project was particularly successful at both solving research problems, and performing extremely well in the competition. For an overview of RoboCup as a whole and the project at UNSW, see [23].

My works in this project is; firstly, continue of works I had done in the summer [22] in learning the walk, and secondly, solving OpenChallenge problem. The environment of this project is rUNSWift RoboCup architecture. However, the current architecture is to use with the old ERS210 dog, and making the architecture works on the new ERS7 robot as well as the old robot had turned out to be an difficult problem (see [22]).

This report presents the works I had undertaken in the project, its problems and solutions, in order to achieve a working architecture that contributed to the outcome of the project. The advantages and limitations of the work will be also described, and future improvements are followed on each chapter.

The report is divided into four main topics: Locomotion, Action-based framework for robot behaviours, The OpenChallenger, and Python development framework, represented in chapter 2,3,4,5 respectively. The main focus in Locomotion is the works on the walking style and walk learning. In order to get the dog learn to walk efficiently, a number of tools were created, which are described in the same chapter. Chapter 3 focuses on a new complete behaviours framework that facilitates solving complex behaviours problem such as the one in OpenChallenger. Chapter 4 is devoted for the OpenChallenge, in which many skills as well as strategies are experimented. Chapter 5 describes the Python framework that helps developing every behaviours of the robots in this year project. Chapter 6 presents two other important tools, apart from SimpleRoboCommander (section 5.3), that are implemented in the system. The conclusion and future works are drawn in the last chapter, chapter 7.

2. Locomotion

2.1. Introduction

This section introduces our implementation of a new walk, Elliptical Walk, along with the learning software module. The learning software is implemented in a generic way in order to plug in different types of algorithms, and different types of search spaces.

With the new ERS7 robot, we first hand tuned the old rectangular walk up to the speed of 27cm/sec, which is as equally fast as the faster ERS210. After realizing the limitation of the rectangular walk, we implemented another type of walk, similar to a related work [6]. ActuatorControl module was also re-organized using Object Orientation so that add new walk styles can be added easily.

EllitiCalWalk was chosen to start off because of two reasons: it is fairly easy to implement and it has a small search space (about 12 parameters, compared to offset walk 24 parameters). Section 2.2.2 describes EllipticalWalk and its implementation. Section 2.2.3 introduces the learning tool, which will be described in length in the rest of this sub-chapter. Section 2.2.4 describes how and why the architecture of the learning software is built, in such a way that it is extensible later on. Section 2.2.5 describes the client in the system, and the implementation of walking robot behaviour. Section 2.2.6 describes how the components interact with each other. Section 2.2.7 describes how to use different learning algorithms by the system. Section 2.2.8 lists a few tools that was developed and section 2.2.9 concludes the topics.

2.2. Background

There are many different walk types that a quadruped robot can have [1]

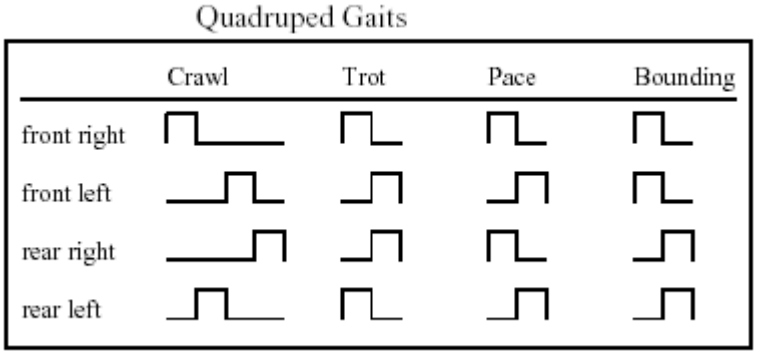

Figure 1 - Different types of quadruped gaits (from 2002 report)

The difference between these walk styles are the synchronization between the four legs. For example, Crawl walk move one leg at a time, pace walk moves left legs in the same manner, so do right legs... Among these, Trot walks are used the most widely, because the front right leg and rear left leg are moving in synchronization, hence making the walk balanced. As a result, the trot walk is shown to be the fastest walk out of the above walk styles. Therefore, in the RoboCup domain, which requires fast walking style, we will try to make a fast trot gaits.

Back to the literature, the first learned walk was done by genetic programming [1]. The technique was used to find a general form of walking, not only constrained to trot gaits. However, the experiment consumed a large amount of time and required modification of the hardware. Over the period from 2000-2002, our team introduced a constraints form of the trot gaits, in which the paws follow a trajectory of certain shapes, for example: rectangular, trapezoidal... [3, 4]. The walk is among the fastest walks in the competition. In 2003, M. S. Kim and W. Uther [5] made a break though by successfully learning the walk automatically. The AIBO walked as fast as it possibly can at the speed of

27cm/sec at the time. The latest work on the field was presented by N. Kohl and P. Stone in [6] to learn the walk in the form of Policy Gradient ReInforcement Learning.

All of these works so far was done on the old robot ERS210. Our goal in this project is to apply them on the new robot ERS7. The new robot has great advantages of faster CPU and stronger motors. Therefore, it is expected to have a much faster walk.

2.3. Elliptical Walk

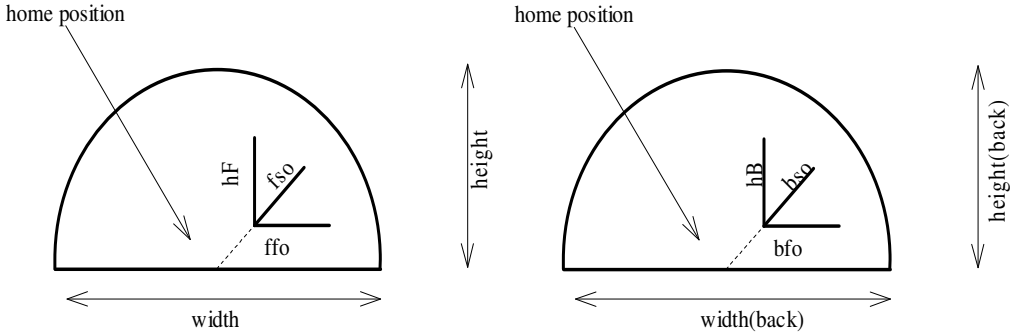

Figure 2 - Elliptical locus of the front and back legs

Ellipse shape: 4 parameters define ellipse width and height, 2 for front legs and 2 for rear legs.

Home position: 6 parameters. These define the coordinate of the home position of the front and back paw in 3D space. They are similar to NormalWalk : ffo, fso, bfo, bso , hF, hB (see picture).

Moving speed (PG): PG (Period ground) defines the number of frames the legs finsh their locus. Large PG moves the leg very slow while small PG slows it down.

Turning adjustment (TA) : This parameter is for auto-calibrating of turning commands. Usually, in order to tell the dog to turn toward a target that has a heading [alpha],the turning command is computed by a formula :

*turnCCW = alpha * TA.*

Where TA is a factor, large TA makes the robot fluctuate about the target, while small TA makes the robot turn slowly toward the target (Figure 3).

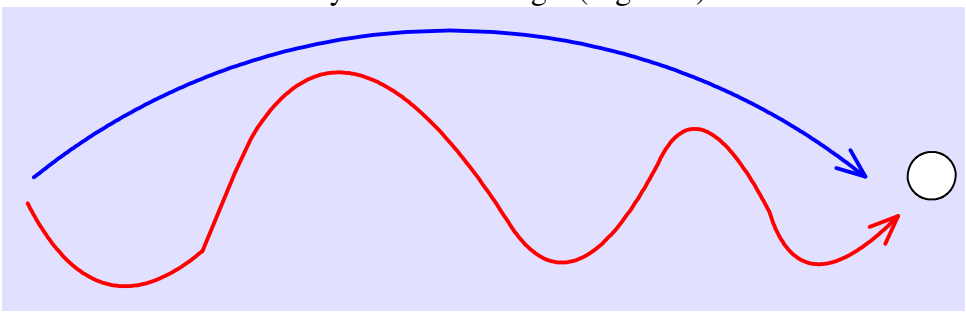

Figure 3 - Impact of turning factor. Blue line corresponds to small factor, Red line corresponds to large factor

The 12 parameters is the first trial to experiment our learning algorithms. The set of parameter is relatively small. Nevertheless, it has a few advantages over the the old Offset Walk. It allows the front legs and back legs to be controlled quite freely by specifying front and back shape differently. This helps hand calibration of the walk. The speed of the leg (PG) can be any integer number while OffsetWalk only accepts divisible by 4 numbers. The turn adjustment parameters can be calibrated by hand easily.

2.4. Walk Learning Software

The software to control the learning process is called Walking Learner Controller (WLC). It can query and display information about the walking robots (Robot Info section), information about the learning algorithm (Learning algorithm section).

Figure 4 - Walking Learner Controller GUI

Each time a walking robot is boot up and starts talking to base station. Its information is displayed in a separate tab in Robot Info section. This is because in each message sent from the robot, the robot number is embedded. Each robot has a learning state (section 2.6), the learning parameters, and last walking time (if it ever runs). The robot is driven by several commands. Each command is executed by pressing the corresponding button. Firstly, the robot is put into "ready" state by "Be Ready!"

command. Next, a "GO!" commands is issued and the robot starts walking forward. A "Skip this run" command tells the robot to stops. The skipping parameter is returned to the parameters evaluation pool (see next section), and becomes available to any other available robots.

There are other commands to deal with exceptional cases. "Run too slow" command is used when it cannot reach the destination in a reasonable amount of time. This is needed because some parameter can make the robot walking on the spot, and unable to move to the target. The robot receives this command will stop and return a big running time to the controller. This can be issued automatically as well.

When the dog crashed, it is treated the same as "Run too slow" situation. However, there is a problem with the client that it also crashed when the dog stops communicating with. When the client crashes, if SimpleRoboCommander tries to send any more packages, there is a bug (in Qt library) that hangs it. Eventhought the WLC support resuming from crashes, it is still undesirable. So a quick way to overcome this is whenever the dog crashes, we have to stop WLC from sending querying messages. Once WLC is stopped, we can then restart the crashed dog and reconnect to all the dogs. That's why the button is needed.

The learning process is started by "Start Learning" button, which starts the learning algorithm and a timer. The timer "fires" every 2 seconds. Each time it fires, it sends a query command to the robot and the robot responses by sending all information about its learning state. The response is then parsed to see if the learning state is "DONE" state. If yes, the walking time is returned to the learning algorithm.

As mention above, WLC can also be resumed from a file. Saving/Restoring functions is actually done in LearningAlgorithm class which is described in the following section.

2.5. Walk Learning Architecture

Walk Learning software is called WalkLearner. It is a module in SimpleRoboCommander. WalkLearner is designed to be flexible and generic. It consists of the following components:

Walk Learning Controller (WLC): this is the main controller and GUI. This is where user starts/stops the learning process. It maintains an Evaluation Pool that generates multiple run from the same policy. WLC queries Evaluation Pool for next evaluation, and send it to RobotInfo to evaluate.

Evaluation Pool: is to generate multiple run from a policy. Each run is evaluated once by some dog. Thus, a policy is evaluated multiple times. By evaluating multiple times, and averaging the result, the experiment is less sensitive to error and more robust. When Evaluation Pool finishes collecting all the evaluations of one policy, it notifies LearningAlgorithm and queries the next policy.

RoboInfo: is the interface between the running dog and the controller. It received command from the controller, encode it and send to the dog. It also receives message from the dog, decode the result, and send back to controller.

Robot: is the walking robot. It is programmed to perform exactly identical to other robot. The only difference is the parameter of the learning walk.

LearningAlgorithm: performs some learning algorithm, for example: Gradient Descent, or Powell Descent Algorithm…

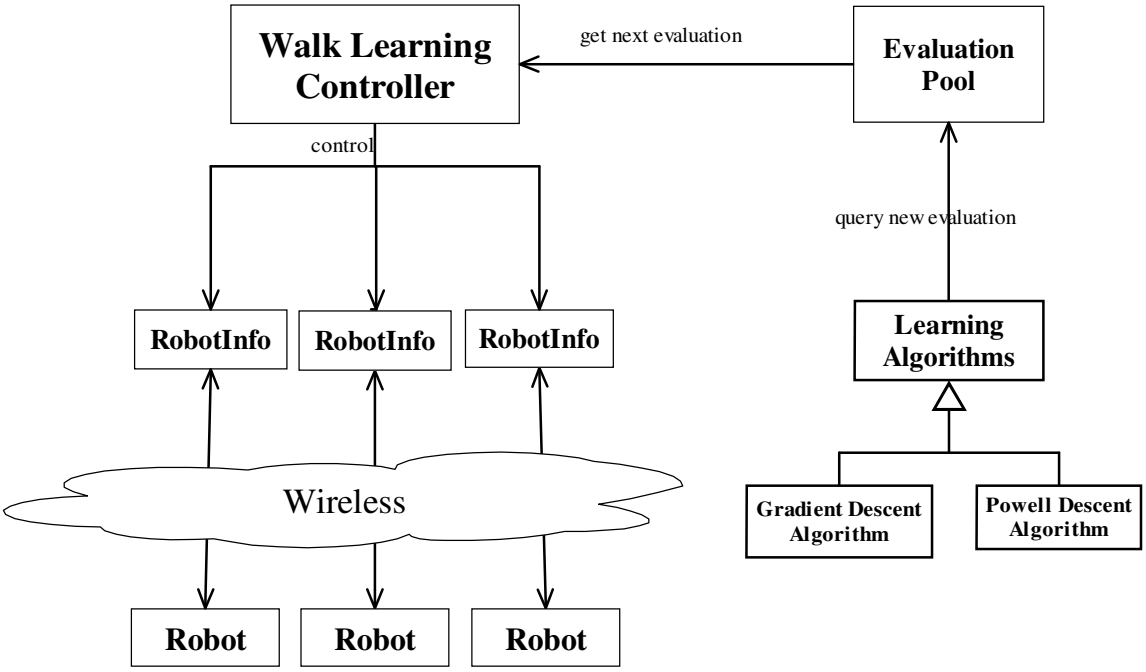

Figure 5 - Walk Learning Architecture

2.6. Implement Walking Learner Robot.

2.6.1. The environment

The goal of the robot is to evaluate a particular walk parameter decided by a learning algorithm. Evaluation is done by having a robot walking back and forth the field measuring the time it takes to do so.

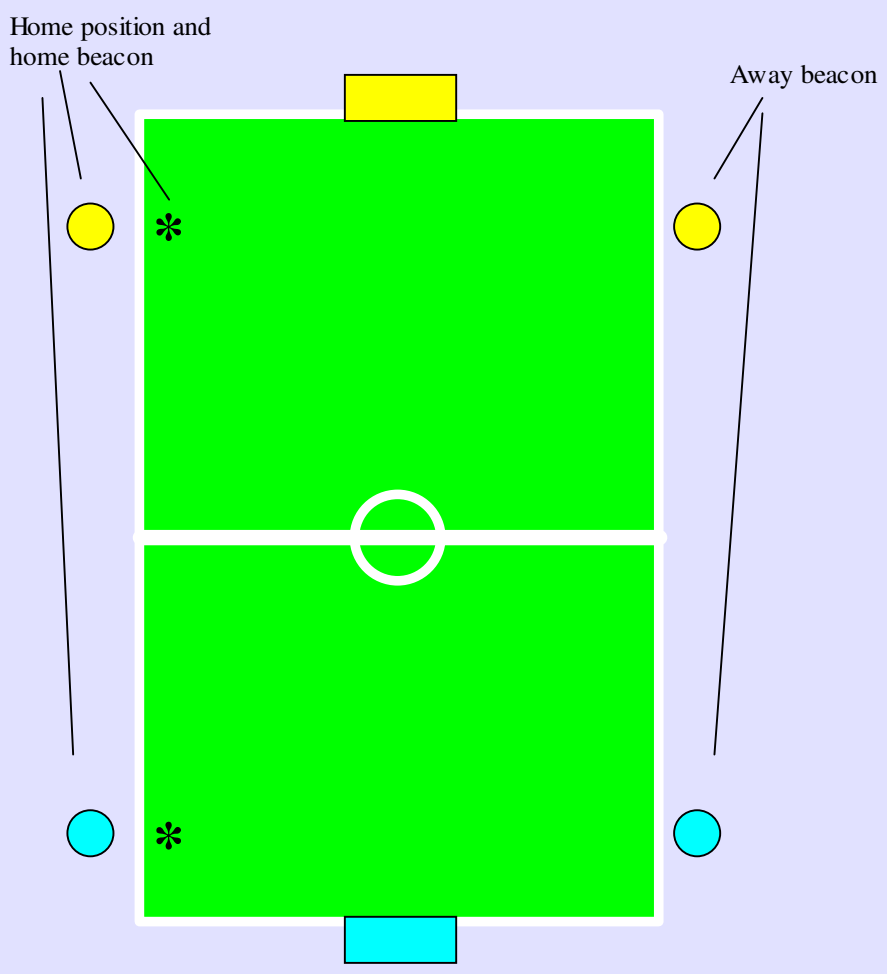

Figure 6 - Walking Learner setup field

2.6.2. The Behaviours

The previous section has explained why we decided to use Polling mechanism to monitor the robot. In order to monitor the robot, WLC needs to know exactly what it is doing. In other words, WLC can only control a number of states and it needs to know exactly which state the robot is in. That is why the robot is designed in a state-oriented way. The below picture shows the states that robot is following.

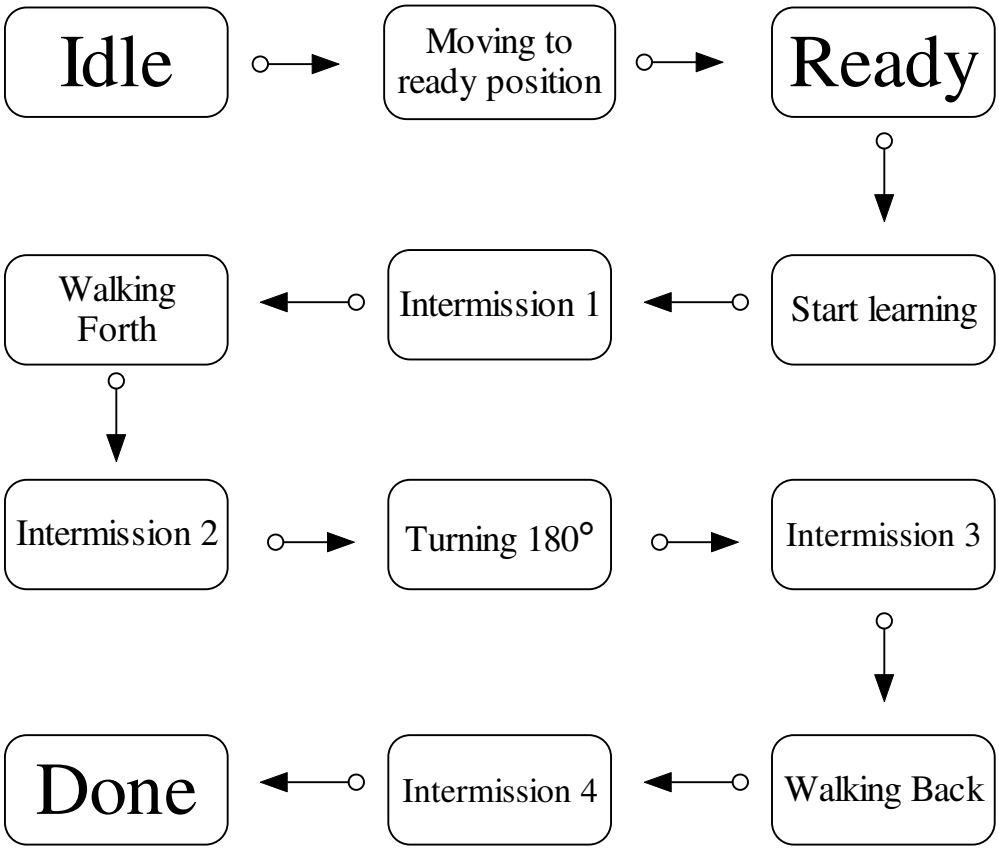

Figure 7 - States in Walking Learner Robot Behaviours

- Idle:
  In Idle state, the dog stays still, does nothing except responding to WLC commands. (Every state responds to WLC Commands).
- Moving to Ready Position:

This state is done usually after the "DONE" state where the robot finishes in front the Home beacon. Therefore it needs to turn around and aligned with the Away beacon.

The dog is to move to the ready position, which is marked in Figure 6. The robot first turns around to find the target beacon. Once it finds it, it steps backward until it hits the field border. When the robot hits the border, the duty cycle value PWM is raised. A threshold of 400 is used to detect such situation. More robust method is available [8] but this method is good enough in this particular situation.

Note that if the dog stops very far away from the Home beacon, it will back off the wrong direction, and move away from the Home beacon. However, it is assumed that before this state, the robot ends up some where near the Home beacon. During the course of the experiment, this assumption is not violated so it is not improved anymore...

Again, this is a design decision and it works. Further improvements is also available that can be taken into account in future work.

- Ready:
  The dog stands still, waiting for next commands
- Start learning:
  This state is in the design, in case some initializations need to be done. However, in the implementation, this just jumps straight to the next state.
- Intermission 1:
  The dog stops for a little while (30 frames). This relax time is needed because PWalk has some delay switching from a walk command from another, unless the dog is currently in standing position. The delay is maximum 1/3 second, which is significant.
  At the end of this state, it starts the timer, and switch to "Walking Forth" state.
- Walking Forth:
  The dog walks toward the Away beacon (see Figure 6), it tracks the beacon. The tracking algorithm is modified because the beacon is not on the ground. Head pan and crane are used to track "in the air" beacon. Pan is equal to the heading to the beacon, and crane is controlled relatively so that the beacon center is in the center of the cplane (Figure 8). The threshold for beacon center elevation is 15°. Delta crane degree is 1°. This means if the beacon is higher than 15° up, more than 15° down the crane is added/subtracted 1°.

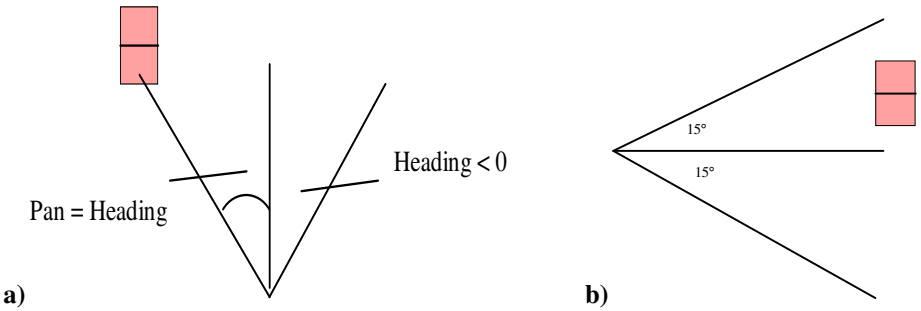

Figure 8 - Head tracking for beacon using a) pan and b) crane

The state ends when the dog gets close enough to the Away beacon. This is checked by the infra-sensor or distance sensor built in the dog.

- Intermission 2:
  This state stop the timer, count the time the dog used to walk from the other beacon (called Tf for time walking forth). Later on when experimenting with this, it is found that the dog often stops at different positions, which causes some error in the measurement. This is due to either delay in PWalk, or delay in the infra-sensor. Therefore the distance from the Away beacon is also taken into account. It is measured again using the infra-sensor. The head pans a range of (-30°, 30°). The distance (Df) is computed as the smallest distance sensed. This is more accurate than the trigger for the dog to stop because the dog stands still and the head moves slower.

How the distance is used to compute to correct the running time is described in DONE state

- Turning 180:
- Intermission 3:
  This state is similar to Intermission 1, which starts the timer for the second run.
- Walking Back:
  This is similar to Walking Forth, only differ the beacon that the dog is running toward.
- Intermission 4:
  This state is similar to Intermission 2, which measures time taken to walk back (Tb), and the distance from the beacon (Db).
- DONE :
  The dog finishes its run on this state; it stops at the Home beacon and waits for the next command.

  When a query is sent, the robot responses to it by a message contains last walking time back and forth the distances from the beacons it stopped at (Tf, Df, Tb, Db).

  The speed of the walk is calculated as (see Figure 9 - Calculate exact speedFigure 9):

$$\frac{FW - Dl - (Df - border)}{Tf} + \frac{FW - Dl - (Df - border) - (Db - border)}{Tb}$$

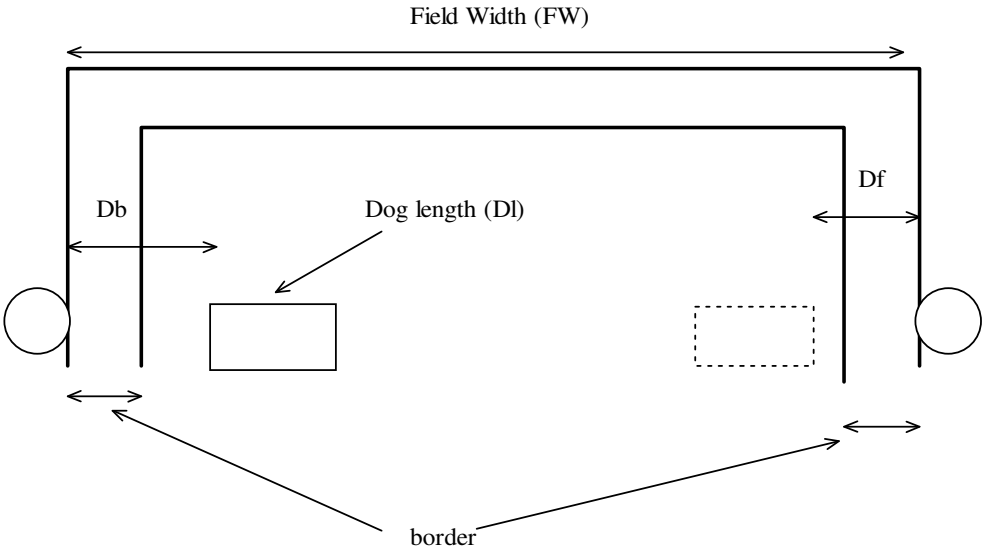

Figure 9 - Calculate exact speed

This behaviours is first implemented in C++, which has its own form of state-based implementation. Later on, after Python development is invented. This is re-written in Python using the "action-based" frame-work that supports State-based as well. As a matter of fact, re-writing the code in Python only took less than one day of work. The Python development and the frame work are described later on in section but it is interesting to compare the two implementations and see how Python and its frame-work help simplifying the later.

2.7. Interaction inside WLC

The whole interaction inside WLC is centered on WLC GUI which mediates between optimization functions (LearningAlgorithm classes) and evaluation functions (robot walking classes). In a typical optimization program, the optimization algorithm manipulates evaluation functions directly. That works very well with mathematical functions or function that is run on the same CPU. On the robot domain, this characteristic is no longer true. The robots are running on its own processor, while the controller is on another computer. In such a distributed system, a number of issues is raised, such as message passing, synchronization, fault tolerance… Therefore it is demanded to have a specialized design. By separating the optimization function and the evaluation function the system becomes loose-coupling, which is easier to manage.

Figure 10 shows how the design works.

- Step 1: WLC GUI initializes a timer. As described in the architecture about the polling model, this timer is to frequently poll the robot.
- Step 2: When the timer times out, it queries all the robots to RobotInfo.
- Step 3: RobotInfo sends query over wireless to robots.
- Step 4: Robot Info receives response from robots as (state, lastRunTime). If the robot has never run before, it returns (-1,-1).
- Step 5: RobotInfo checks if the state is DONE, then requests new command (new parameters) to WLC GUI. It also notifies WLC GUI about the pair (lastRunID, lastRunTime) in the request message. runID is to differentiate between different runs.
- Step 6: WLC GUI, in turn, query Evaluation Pool to see if there is available run.

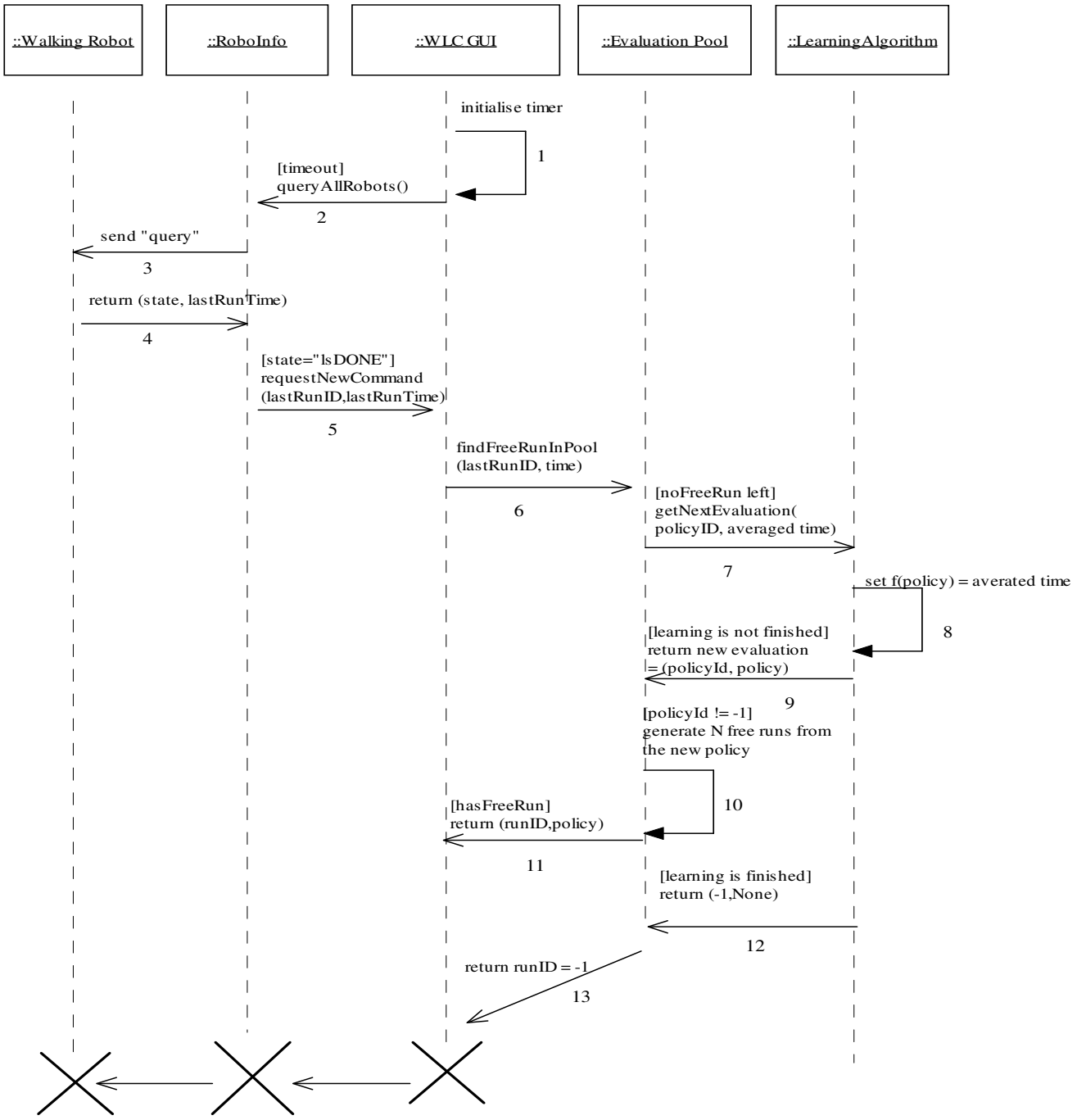

Figure 10 – Walking Learner Sequence diagram

- Step 7: Evaluation Pool receives lastRunID,lastRunTime. It knows which runID matches which policyID. If policyID has been evaluated for N times (N is the number of times a policy is evaluated in order to get reliable measurement), it returns a reliable average time to LearningAlgorithms. Otherwise, the same policy with different runID is returned for the robot to run again.

- Step 8, 9: LearningAlgorithms gets the evaluation, work out the next parameter to evaluate and return the pair (policyID, policy) to evaluation Pool.
- Step 10, 11: Evaluation Pool receives new policy; it generates N runs, put into a queue and continuously returns to robots to run.
- Step 12, 13: If LearningAlgorithms finishes, it returns (-1, None) so that WLC GUI can stop polling and terminate.

2.8. Implementing LearningAlgorithms interface

2.8.1. The interface

The system is designed to easily add new algorithms by simply extending the LearningAlgorithm class. It involves implementing the following methods:

- __init__(startFromScratch, logFileName): initialize the algorithm. It takes 2 inputs; startFromScratch specifies whether it should resume from previous session or start over again. logFileName is to load/store session data.
- setParameters(…): set initial parameters.
- getCurrentParameters(): return what is set in setParameters.
- getNextEvaluation(): return (policyID, policy), the next policy to evaluate. The algorithm remembers that this policyID is being evaluated, and wait for it to be set by the function:
- setEvaluation(policyID, value): set the evaluated value for a policy identified by policyID.
- getBestParameters(): return the best policy so far.

The rationale of having getNextEvaluation() and setEvaluation() is that the evaluation can be done in parallel. Some algorithm can be done in parallel (for ex: Gradient descent) and some cannot.

2.8.2. The adapter

While the new system is more convenient to handle distributed evaluation, for some algorithms, it is harder to implement. This section will present how we adapted a traditional optimization function to LearningAlgorithm interface. With this adapter, any existing optimization function can be "plug-in" into our system without re-implementation.

Powell optimization will be used as an example. The algorithm is implemented in a function called fminPowell (in powell.py file). The adapter is implemented in PowellDescent.py

We can think of fminPowell as being active, while LearningAlgorithm interface serves as a passive object. One way of solving this problem is using threads.

The adapter essential invokes another thread to run fminPowell, and the main thread will handle LearningAlgorithm interface.

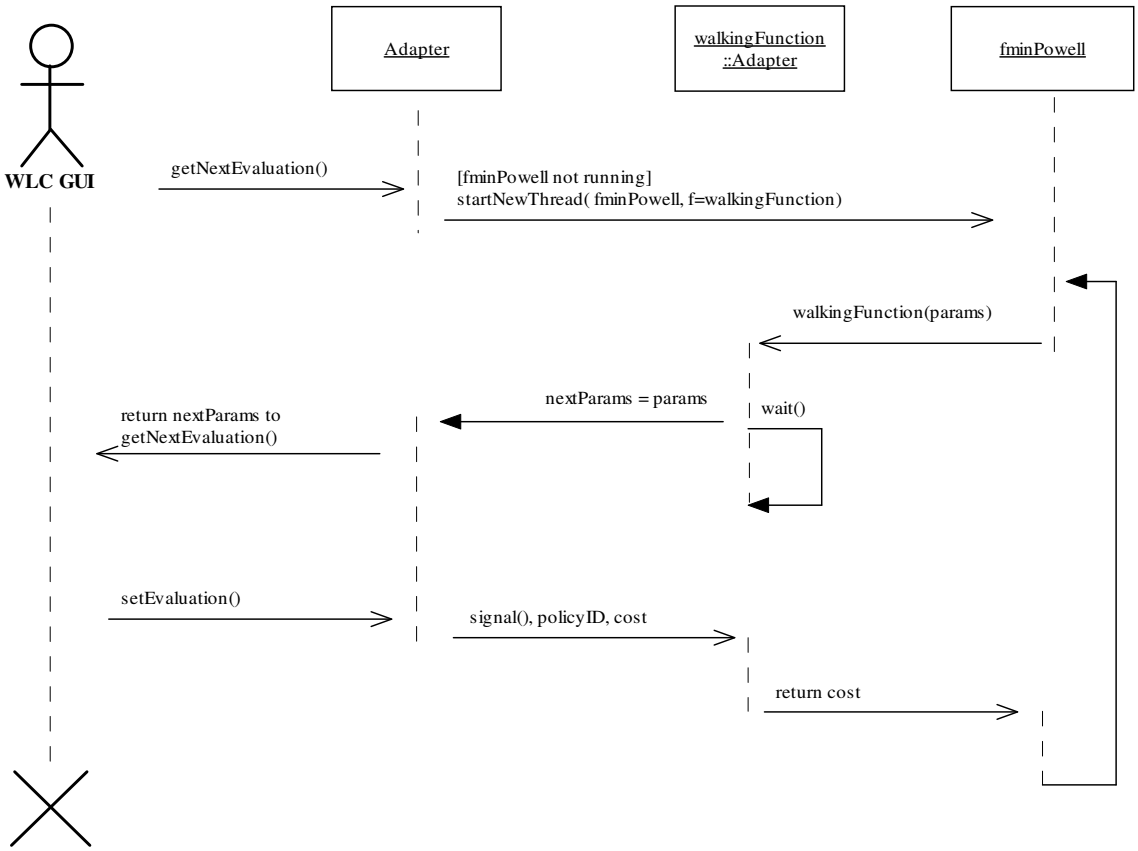

Figure 11 - LearningAlgorithm Adapter sequence diagram

The adapter provides walkingFunction method, which is a pseudo evaluation function. What it does is getting parameters from fminPowell, put it into evaluation queue, and wait until setEvaluation is called. This function gets called from fminPowell, hence it is in the same thread as fminPowell.

When getNextEvaluation is called, it wakes fminPowell up and waits. fminPowell runs and eventually makes a call to its evaluation function (which is walkingFunction), walkingFunction puts the parameter in the queue, wakes Adapter thread up, and waits.

When setEvaluation is called, it has the result that walkingFunction is waiting for. It signals walkingFunction up, hence resumes fminPowell threads. fminPowell then continues the cycle, runs and makes call to walkingFunction…

In conclusion, the adapter shows the extensibility of the current system. It is possible to implement any algorithm, or use existing algorithm library, to experiment with the dog.

2.9. Walk Learning subsidiary tools.

- Function variance measuring tools

To test the accuracy of the walking robots, we have the dog run the same policy a number of times, and get the mean and variance. A simple tool to do the work was implemented as a LearningAlgorithm, called LearnError.py. It simply returns the same parameters in getNextEvaluation() and records all values given by setEvaluation(). In the end, it prints out all statistics data, such as mean, variance…

The tool helps reducing a number of bugs in the walking robot. We kept modifying the behaviour until it can measure the time with a small enough variance.

- Testing algorithm offline

Before the two learning algorithms are run on the dog, they were tested offline by evaluating some known function. This is to make sure about the correctness of the algorithms. The well-known function Rosenbrock was used, and the algorithms were shown to converge.

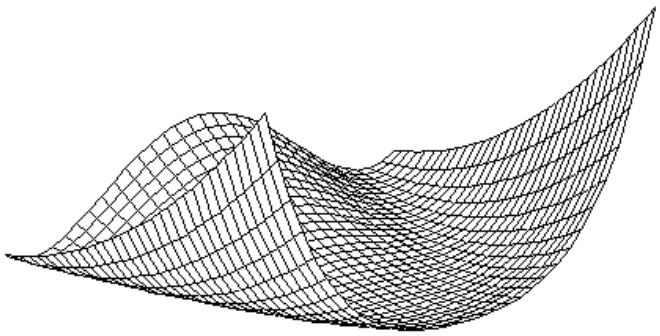

Figure 12 -3D Visualization of 2 dimemsional Rosenbrock function. (The opitmizer starts from the left hand side corner and has to reach the optimum on the right hand side corner)

2.10. Conclusion and Future Improvement

The learning frame work was used to improve EllipticalWalk. A reasonably fast walk was found in only one hour. The speed is about 34cm/sec, which is significantly faster than the old dogs. The team's focus was switched to improving other modules. Therefore, no other improvement was made since then. Many ideas have to be put into future works.

Several more sophisticated walking styles were experimented, such as SkiWalk, SkiEllipticalwalk. These walking styles have a large number of

parameters, allowing more control of the way the legs move. An example of that is timing of the loci. In all other walks that we had implemented so far, time is allocated evenly along the trajectory. Such timing influences largely on the legs' synchronization, which largely affects the walk performance. Another extension that SkiEllipticalWalk has made was being able to adjust the center of gravity of the dog. In another word, it can make the dog more balance while walking. Such consideration might have direct effect on the walking speed as well. Adding more parameters, and let the algorithms figure out the best parameters would definitely increase the speed.

Up to now, most effort had been spent to learn the walking straight walking style. However, in a game situation, not all the time, the dog walks straight forward. A good walking dog should be able to back-off, to get behind ball/team mates, to walk side way…efficiently. Such skills usually need a lot of fine-tuning. In addition, at the competition, it is viable that some dog walks diagonally faster that it does straight. Getting the frame work to learn those walks will certainly increase the dogs' performance in games and is very desirable in the future time. In fact, with the frame-work, it is quite simple to add such features.

When the parameter space gets large, searching might takes a large amount of time. In such situation, special knowledge might be taken into account to increase the search strategy. One example of special knowledge was introduced above, which is the balance of the dog, which can be adjusted separately. Another piece of knowledge is that the front legs and back legs can be separated. The rationale of doing this is that the set of parameters for the front legs seems to have little effects on the parameters of the back legs themselves. Therefore we can improve the back legs by just changing parameters for the back legs. Then we can improve the front legs the same manner, then alternatively optimize back legs, front legs. The idea is similar to co-evolution in genetic programming [9]. In fact, when we do manual fine-tuning of the walk, a lot of special knowledge (like balance, contact point with the floor, synchronization…) is considered. Such methods are worthwhile to research on in the future.

Normal walk (ERS210)	24cm/sec
Zoidal walk (ERS210)	25cm/sec
Offset walk (ERS210)	27cm/sec
Normal walk ERS7	27cm/sec
Elliptical walk (ERS7)	34cm/sec

Figure 13 - Performance comparison table of different walks

The result of the whole described work is a reasonably fast EllipticalWalk. The speed measured in the lab is 34cm/sec. The performance can be obviously improved since there are many room for improvement addressed above.

3. Action-based frame work for robot behaviors

3.1. Introduction

In the past, writing behaviors for the AIBO usually takes alot of time. This is due to the fact that debugging code in robotics environment is very time-consuming. In fact, there has not existed a concrete frame work that supports developing robot's behaviors. This section introduces an "action based" frame work that helps developing complex behaviors. Developing behaviors consists of writing, debugging and maintaining code. This frame-work is developed with debugging and code-reusability in mind.

The frame-work has also tried to incorporate many techniques in writing robot's behaviors, such as hysteresis, time orientation, decision-tree based, state-based,…The frame work is written in python, however, it takes advantages mostly from Object Oriented Programming rather than Python features. Therefore, it can be written in any OO language, such as C++ or Java.

This frame work has been experimented to implement a number of robot behaviours including OpenChallenger (chapter 4). Other behaviours paradigms, such as Walking Learner Robot (state-based behaviour, section 2.6) and Odometry learner (time-based behaviours, Appendix 2) was also implemented using this frame work.

Section 3.1 will survey the back ground. The details specification is described in section 3.2. Section 3.3 provides an insight comparison of this frame work with traditional state-based and decision-tree based frame work.

3.2. Background

Behaviour is one of the key areas to contribute to any team success in RoboCup competition. This is the area where all the great researches in other area can be used, and be proven to the public. In the past two approaches to writing behaviour were used:

- Decision-tree based:

  A decision tree is used to decide the execution path of the code. From the top to the leaf, a decision is made at each node, typically in a structure as depicted below (extracted from 2003 report).

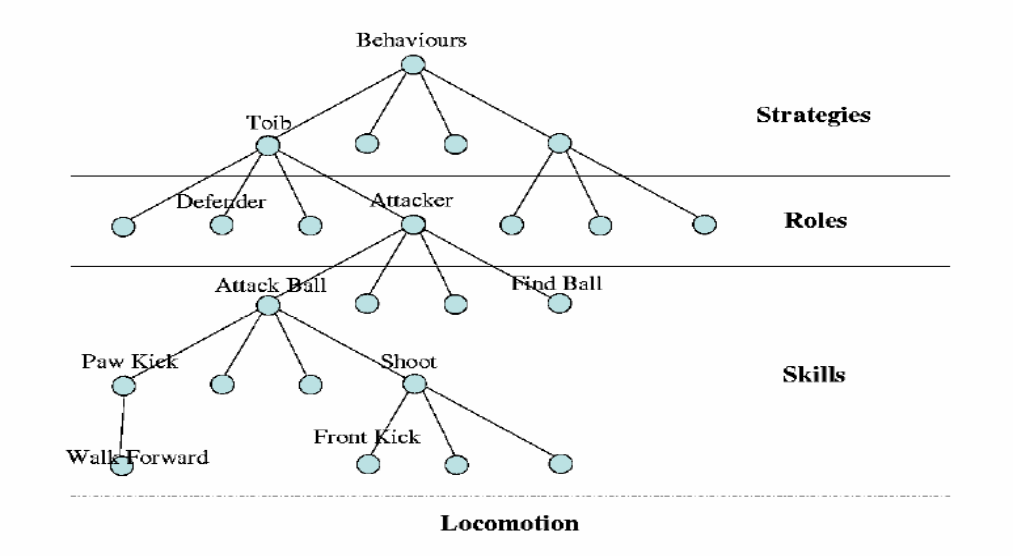

Figure 14 - Decision-tree based behaviour

The advantage of this method is the simplicity of design, because any decision tree can be coded by single statement "if …then…else…" Almost any language or programming paradigm would have supported the statement, thus, decision tree can be programmed.

The difficulty of using this approach is not small. It is usually hard to create a decision tree because careful decision, hysteresis (see 3.3.5) need to be considered, to make sure a particular situation is not falling into two different execution paths. There is not interaction between nodes in the tree. There is no history being stored about the past decision, for example whether a particular action is achieved or failed... these kinds of information are sometimes needed to make the right decision. Another problem with decision tree is that modifying the tree is generally cumbersome, because usually, a decision at one node is strongly dependent of all the decision made above it. Hence moving a node under different often easily creates undesired behaviours.

Another problem with decision tree is that it is unclear how to plan behaviours over time. On the other hand, RoboCup world is actually real time. Therefore, a notion of time should be considered in any approach.

- State based:

An alternative of decision-tree is state based method. Each decision is now associated with a particular state, which enhances maintainability, because decision inside one state does not need to care about decision in other state. In another word, a decision depends on one only thing, which is the state it is in.

An example of states in a simple behaviour of the goalie which is used by the German team (extracted from XABSL paper [24]):

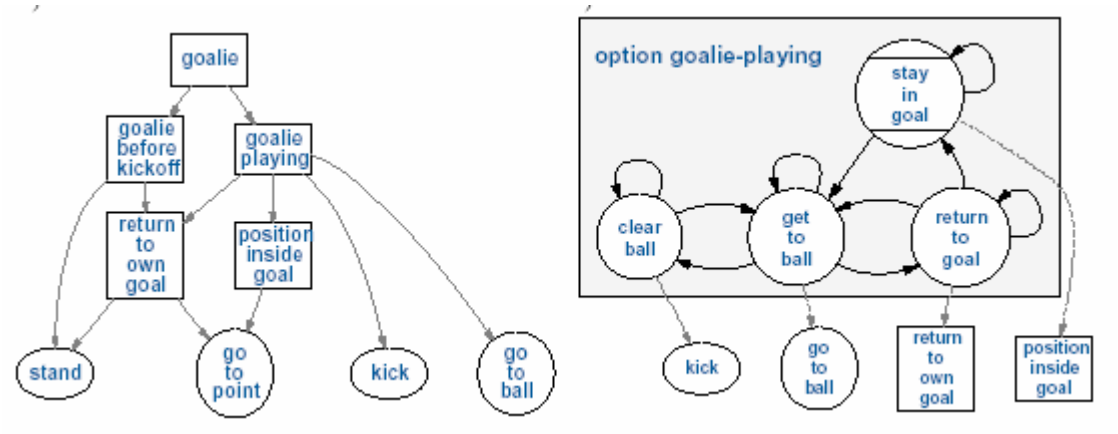

Figure 15 - Goalie behavioural states

The draw back of state based is that even though decision inside a state is independent of outside world, there are still decisions to switch from one state to another state. In the paper, such decisions belong to the former state. Therefore, a state is still dependent too much on other states, which complicates the design.

In practice, there are many variances of these two methods. In 2003, our team used a combined solution by keeping the main decision tree, however, introduced lock mode, which is some kind of states the an action belongs to.

One of the successes of German's XABSL language is that it is a complete specification of the robot behaviour. It helps increasing team's communication because team member only need to understand its specification language, not other's code.

3.3. Specification

3.3.1. Overview

Any type of an agent's behaviours can be seen as the figure below

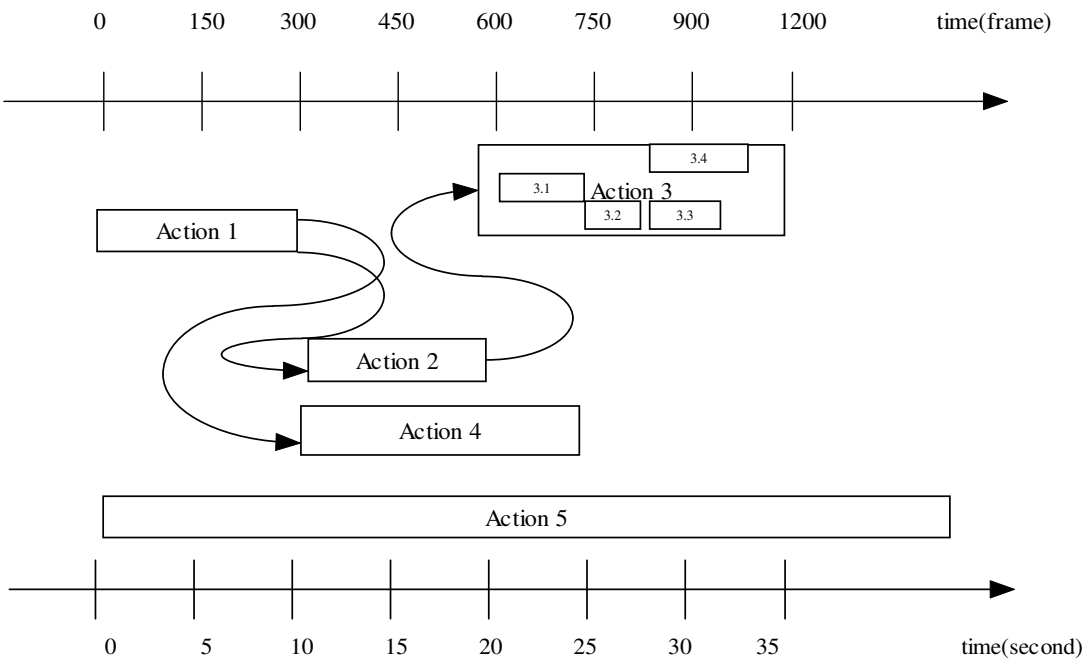

Figure 16 - A representation of behaviours

The agent is executing a number of related behaviours, that we call Action. Some actions are executed one after another (action 1, 2 and 3), while some actions are executed in parallel (action 1 and 5). If we zoom in Action 3, it is composed of many other smaller actions, interacting with each other the same way its parents do. The transition from one Action to another is represented by an arrow. At one end of the arrow, usually some decisions are made, to decide with arrow to follow…

In this section, we will present a framework that depicts almost every thing that happens in the above agent's world. Section 3.3.2 describes in details the notion of an action that is mentioned above. Section 3.3.3 shows how to write code for an action. Section 3.3.4 and 3.3.5 show another two aspects of the frame work: the time counter and hysteresis. Section 3.3.6 describes how behaviours can be extended. Section 3.3.7 shows a common way of connect the actions together, by "pipelining".

3.3.2. Notion of Action

Definition: An action is a series of decisions that are made in consecutive number of frame in order to archive something.

For example, if a "running" is executed in frame 12,13,14 then it is an action "running", while if it is executed in frame 12,13, 15,16 then it is considered as two "running" action.

The reason that an action needs to be executed in consecutive frames is because of the way behaviour is designed to send action to actuator control. At each frame, if behaviour does not set the atomic action, a default action will be taken, for example: freeze the dog. Therefore, if the dog stops doing action for one frame, all the effects that it did in the previous frame would be lost, and the dogs has to start acting from the beginning.

3.3.3. Writing code for an Action

Many interesting events usually occur at the start and ends of an action. For example, after dribbling the ball, kick it; open the mouth before grabbing ball, close it when finish… It is important to be notified about these events.

The frame work provides two methods to handle the events of starting and ending an action:

- beginAction() is executed at the starting frame, just before DecideNextAction()
- finishAction() is executed at the ending frame, just after DecideNextAction(), or executed one frame after the true ending frame. (When "late finish" occurs).

With the ability to program the beginning and the end of an action, the robot is able to do something like: open mouth, turning (take about 30 frames), and close mouth when finishes (TurnKick).

The code that does this looks like (extracted from KickTurning action, OpenChallenger.py):

```
def beginAction(self):
                …
        HelpLong.openMouth (GRABBING_MOUTH_OPEN)

def finishAction(self):
        HelpLong.openMouth  (Constant.IND_MOUTH_CLOSED)
```

Figure 17 - beginAction() and finishAction() example

Note that the finishAction() may be called in the next frame, because the action might not intentionally ends, but being forced to end by a higher decision. Therefore, a "late finish" may occur; however, it is always finished before any other action is begun. This does not create any known problem so far.

3.3.4. Built-in counter

In order to facilitate time oriented behaviors, a frame counter is available in each action. The counter's value is the number of frames that the action has been executed. It is increased every time DecideNextAction() is called.

The counter information is useful in any kind of action that must be aware of time. For example, in GrabBall functions (see section 4.3.1.2); a chin sensor check can be implemented as:

```
#check chin sensors
 if self.counter > MyGrabBall.CAN_CHECK_CHIN_TIME :
        if VisionLink.getAnySensor(Constant.ssMOUTH) -
        GRABBING_MOUTH_OPEN >
        MyGrabBall.STUCK_THRESHOLD:
                print "mouth sensor"
                return self.outState
else:
       return None
```

Figure 18 - Example of returning states

This checks if the current angle of the mouth is far less than the angle that we tells the mouth to open. If yes, that can only mean there something (ball) gets stuck under the chin, so the robot knows that "ball is under chin" and hand itself to the next action (by "return self.outState"). The time counter check here is necessary because, it takes time for the mouth to fully open, the amount of time is specifed in MyGrabBall.CAN_CHECK_CHIN_TIME. If that amount of time is too short, the mouth is still closed, and the check would always return true, which is not correct.

While 'counter' tells the local point of time, another piece of information is also available in the framework that is the value of the global counter when the action is last executed. The value is stored in member variable "lastFrameID". This is useful in some situation for example: when is the last time robot does active localization (action "Active localization" is called…). However, this feature is never used in practice.

3.3.5. Hysteresis

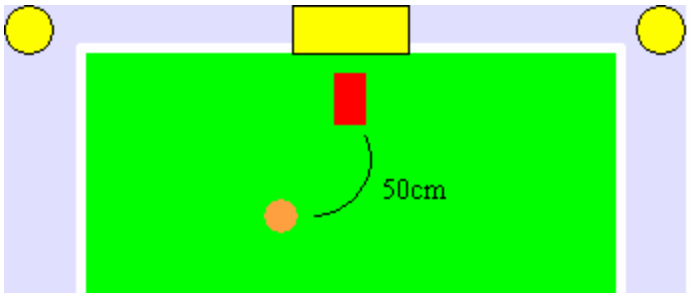

Figure 19 - Goalie attacking ball 50cm away

One common problem when writing robot behaviors is to make the right decision. For example, the goalie has to decide to either attack the ball or stay the same to block the goal. A simple solution would be

```
If [the distance from ball] > 50cm then
        Attack the ball
Else
        Stay at the goal
```

The main flaw of this problem is that because the ball distance is never be the same even the robot stays still. It usually fluctuates within some range around the true distance. Assume when the robot is 50 cm away from the ball, the distance fluatuate from 49cm-51cm. What may happen is when the robot is away from the goal box as in the picture, the ball is 50cm away from the robot. The distance calculated is 49 cm. so based from the code, the robot decides to attack the ball, however, in the very next frame, the calculated distance becomes 51 cm, and the robot decides to go back to goal, then in the next frame, the calculated distance says 49 again and the robot attack the ball again. So the result of this is that the robot will hang around the spot without doing anything, and give chance to opponent robots to get the ball.

This problem can be solved by hysteresis.

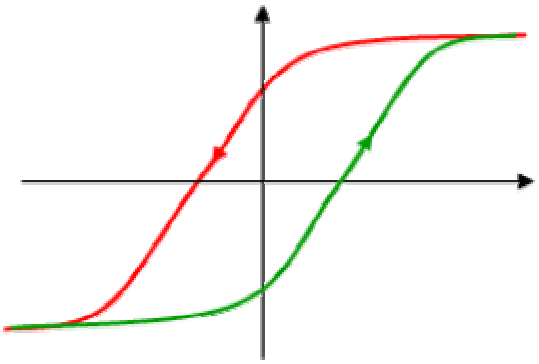

Figure 20 - Hysteresis curve

"Hysteresis is a property of physical systems that do not instantly follow the forces applied to them, but react slowly, or do not return completely to their original state: that is, systems whose states depend on their immediate history" ,Wikipedia.

Imagine the force is the sensor information that the robot senses, the system is the action that the robot is doing. In some situations, the robot should not react immediately to the sensor. In our example above, it depends on the previous state (immediate history) that the goalie reacts to the ball distance differently. If it is attacking the ball, only when the ball distance > 70 cm, it can change its mind, to go back to protect the goal. When it's protecting the goal, it can only attack the ball when the distance < 30cm. In another words, the condition to switch decision is made harder, hence enable smoother robot behaviors.

To enable hysteresis between 2 actions, say AttackingBall and DefendGoal, we need to define 2 conditions, one to switch from AttackingBall to DefendGoal B and another on to switch from DefendGoal to AttackingBall.

There are two ways hysteresis can be implemented in the frame work. The first way is in child actions and the second way is in parent action. The fisrt soluion is pretty straightforward from out definition of hysteresis above.It contains defining an only function called switchingCondition() that returns True if the action need to be switched and False otherwise.

In AttachingBall class we have

```
switchingCondition()
    return distanceFromBall < 30
```

In DefendingGoal class we have

```
switchingCondition()
    return distanceFromBall > 70
```

Now, we need a generic function to do hysteresis, we call the function

```
doHysteresis(currentAction, action1, action2):
        if currentAction.switchingCondition() == True then
                if currentAction == action1
                        currentAction = action2
                else
                        currentAction = action1
        else
                currentAction is unchanged

        do current Action
```

The goalie code would simply have

```
When Initialisation (beginAction()) :
        currentAction = DefendingGoal /* default action */
In DecideNextAction() (is executed every frame):
        doHysteresis(currentAction, AttackingBall, DefendingGoal)
```

However, this solution has 2 flaws: The first one is that this results in scattered code. We put write 2 functions at 2 different places to achieve the same purpose. That makes the code less readable and harder to maintain. The second flaw is that if we want to do hysteresis between AttackingBoal and another action but not DefendingGoal, then it is impossible, because the condition of switching action is hard coded in AttackingGoal, which is meant to be used in conjunction with DefendingGoal.

Therefore we came to conclusion that in order for the action to be flexible and portable, the child action should not know about what hysteresis it is used with. Moreover, the hysteresis belongs to the parent class which is manipulating the child actions. Hence, hysteresis should be implemented in parent class. Fortunately, the parent class always keeps the pointer to which child it is using. We have hysteresis function:

```
doHysteresis(action1, condition1, action2, condition2)
        if this.subState == action1 then
                if condition1() == True:
                        this.subState = action2
        else
                if condition2() == True:
                        this.subState = action1
```

The function relies on 2 callback functions: condition1() and condition2() (which essentially the same as AttackingBall.switchingCondition() and

DefendingGoal.switchingCondition() mentioned above). Fortunately, most language supports callback functions, so this solution is quite feasible.

This solution is portable because the child actions themselves don't need modifying. It is also easier to maintain because all the aspects of hysteresis is well-defined. They are defined in the same chunk of code. So anyone who is unfamiliar with the code but understand hysteresis can comprehend easily.

That is the case of bi-state hysteresis. Multi-state hysteresis can also be done the by decomposing into bi-state hysteresis. However, this is not a common design so it will not be described in details here.

3.3.6. Extensibility of an Action: by Inheritance and by Composition

Extensibility is one of the main design goals of the framework. It makes the system more reusable. New features or behaviours are added into the system by re-using existence Action, instead of re-writing from scratch. That makes the existence code more robust because it is not only used, but also the base of other code, which adds another chance of testing to the original one.

The two ways of extending an Action is by Inheritance and by Composition. These concepts resemble those in OO Programming. Furthermore, they can be implemented using OOP.

By Inheritance, an action is made a subclass of a parent action. All the methods as well as attribute of the parent action are inherited. Usually, we only want to inherit the general behaviours, and specialize some specific details belongs only to the inheriting action. Therefore, most often, the method DecideNextAction() is inherited, and the child action provides some new methods to set attributes that are used in DecideNextAction(). For example, suppose there is an Action GrabTurning, which grabs the ball, hold it and turn by a desired angle (see 4.3.1.4). We want to create another action, which grab the ball, turn toward a teammate, and kick it at the end of the action. GrabTurning has an attribute desiredHeading, which is used in DecideNextAction () to turn by. We know that at the begin of each action, beginAction() is called, and at the end of each action, finishAction() is called. Suppose the new Action is called PassingToFriend. Instead of rewrite the whole action from scatch, it can be extended from GrabTurning by just re-implementing beginAction() and finishAction().

```
PassingToFriend.beginAction():
    desiredHeading =
    friend_to_me 's  heading – my
    own heading
```

```
PassingToFriend.finishAction():
    setKickToUse( ForwardKick)
    setForceStepComplete() // this
force ActuatorControl to the the
kick immediately, no delay.
```

Figure 21 - PassingToFriend's beginAction() and finishAction()

Another way of extending action is by composition. This concept is more natural to think of, and is used in most of the skills in section 4.3 (OpenChallenger). Composition is something composed or made of something else. In the next chapter, we will be seeing RunToGrabBall (4.3.1.3) action composes of HoverToBall, GetBehindBall, GrabBall, and ApproachingBall… In general, whenver we have a high level behaviour, which makes use of several lower behaviours, we can use Composition. In the high level action (RunToGrabBall), the lowerer actions (HoverToBall, GrabBall…) are stored as (private) attributes. The higher action can either actively decide which lower action to use in each action frame, or passively "pipe" lower actions together using Dynamic Architecture (3.3.7).

3.3.7. Dynamic Architecture by pipelining

One of the aims having the new framework is to develop behaviors that is flexible and can be reused easily. Usually we want to write behaviors such as "Grab ball then kick the ball" as well as "Grab ball then dribble the ball". It is desired to factor "GrabBall" action in these two behaviors. This requires actions to be self-contained and a way to "bring" the actions together.

The first requirement is addressed and solved by the idea of action in section 3.3.2 above. For the second requirement, there are a number of ways to solve. The German invented a XML-based language to specify how states are connected together. As discussed in 3.1, this results in a complex structure and potentially an enormous number of connections. In order to keep things simple, we only needs to know which action is executed after another action, hence assimilate a pipeline structure. The idea is originated from ArchJava which is in turned inspired by Unix's pipeline mechanism.

In Unix, each shell program is designed to do a relatively simple task. It follows a standard that each program read input from stdin (standard input) and writes output to stdout (standard output). Complex tasks are then achieved by pipelining programs one after another. For example we can do "ls | sort " (list the content of directory then sort it) or "ls | wc –c" (list the content of directory then count how many lines)…

Back to the proposed framework, each action has a pointer to the next action, called "outState" (it is originally influenced by the state-based framework). In order to do action, say A, after another action, say B, we simply set the outState pointer of A to B.

A.setOutState(B).

We can query the architecture which action is executed after which action:

print A.getOutState(B).

We can also "disconnect" actions by:

A.setOutState(None).

Since connecting and disconnecting actions can be done in anytime. The architecture can be dynamic. Two common uses are Static Architecture (SA) and Dynamic Architecture (DA). SA means connection of actions is done once and only once in initialization time, hence the architecture is unchanged overtime. DA means connections are done in runtime. An example of DA is demonstrated in BallHoldingLocalise action (Appendix 1)

3.4. Conclusion and future improvements

The framework has addressed a number of problems with both old methods of writing behaviours. With decision tree, it is difficult to model low level skills. The framework solved it because by low level action, which can be written in a time-based paradigm. With state-based method, complex strategies usually involved too many states, which is too hard to manage. The frame work managed that by

The framework has been used and shown many advantages in developing the behaviours. It helps writing "plug-and-play" code. The action is written once, and can be used anywhere in other actions. It helps testing, since each of the action itself is a complete, run able behaviour. The framework has tried to apply a well-known concept in software engineering: separation of concern [7], which makes the code written in a more manageable and maintainable way.

One issue with this framework is performance. It is written totally in Python. Therefore, one improvement would probably involve embed the whole framework in C++, which will definitely make it faster.

Having got the framework, it is possible to provide more tools that support the framework. A monitoring/logging tool would be certainly helpful in analyzing the strategies. The log is stored, or sent over to base station, where a certain interesting pattern can be analysed. For example, "an action X is never executed" may imply that something wrong may have happened. Tool to detect anomalies can also be developed to detect something like: the action Y is suddenly executed where action Z is normally executed…

4. The Open Challenge

4.1. Introduction

"This challenge is designed to encourage creativity within the Legged League, allowing teams to demonstrate interesting research into autonomous systems." Robocup committee.

The Open Challenger is a new challenge of this year's competition. As the name suggested, it is open to any sort of research ideas, and more importantly, it must be challenging idea.

The other two challenges that the team have this year are Vision challenge (Variable Lightning Challenger) and Localization challenge (Almost SLAM Challenge). With the aim to make the challenges complete, we decided to have a challenge in high level behaviours, intelligent strategies and multi-agent cooperation. The challenge is expressed in the form of Keep Away Soccer game. Keepaway soccer is a challenging enough task for the following reasons:

- Two dog needs to cooperate well in order to play keepaway.
- Low level behaviours and skills involved in keepaway soccer may even more subtle than normal soccer, including dribbling, catching ball…
- The information of the environment is stochastic and not complete, the dog may not know exactly where the opponent is, or teammate is.

On the other hand, Keepaway Soccer is, in fact, a well-known problem in machine learning. The simulation of the game can be learned to play pretty well by ReInforcement Learning and Genetic Programming [10, 12]. However, it has never been experimented on a real world condition as in robocup. That makes the problem very interesting to tackle.

During the course of researching on this challenge, we have found it much more involved than any previous challenges. The challenge requires solving many sub-problems related to vision, localization, planning, communication…

Even though the work was still in progress and did not perform successfully in the competition, we believed that it has established a quite strong foundation for potential subsequent works.

Section 4.2 presents the very high level strategies. Section 4.3 described the skills in more details, including low-level actions and higher level decision trees. Section 4.4 concludes the chapter and sum up the future works. The content of this chapter is closely related to the Behaviours framework described in previous chapter, because the KeepAway soccer not only contributed many of the ideas into the framework, but also took advantage of those concepts from it from the very beginning.

4.2. The big picture

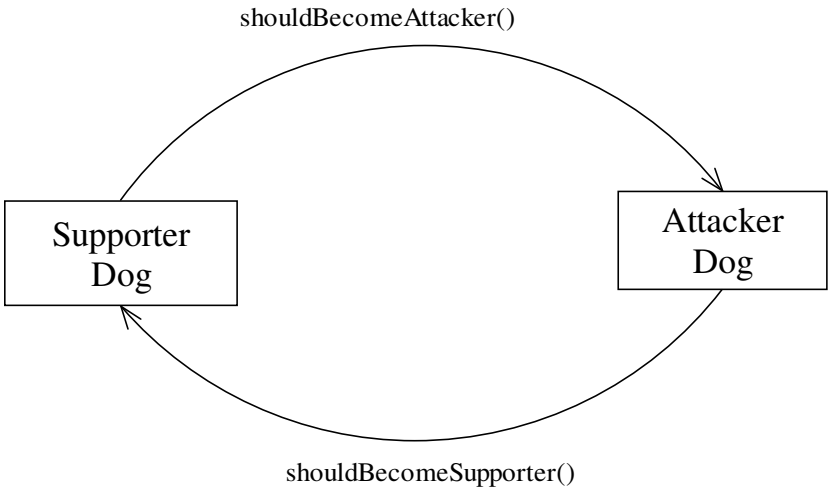

Figure 22 - Two roles in the OpenChallenger

There are attacker and supporter. The idea is to balance between the two roles. The roles are described in following section 4.2.1 and 4.2.2. The role switching algorithm is described in section 4.4.1.

4.2.1. The Attacker

The main role of the attacker is to either handle the ball, holding it, passing it or keeping it away from the opponent. Since the attacker is busy handling the ball, it relies on its teammate to locate the opponent, and use the information to plan the actions.

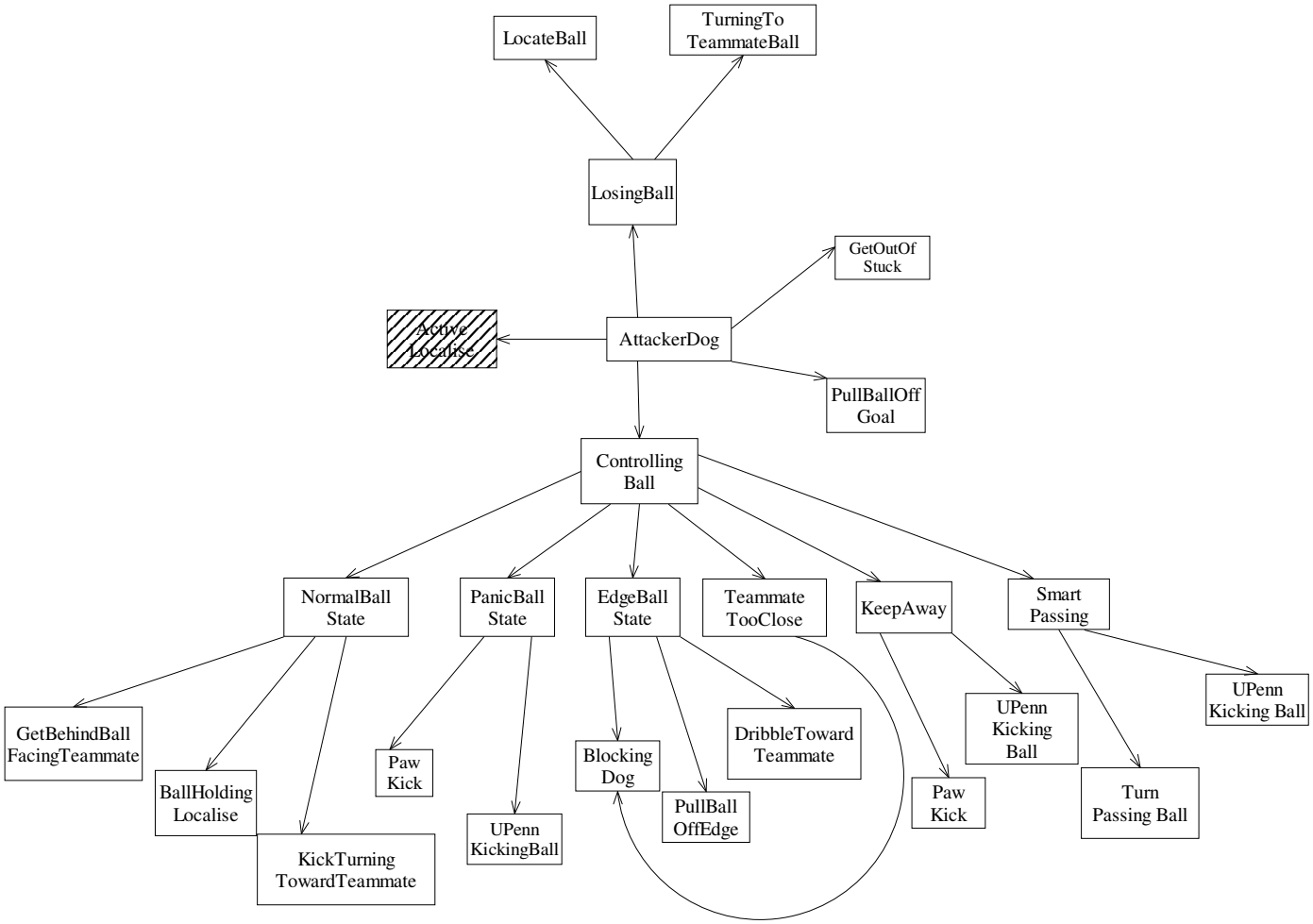

Figure 23 - Attacker Action Tree

4.2.2. The Supporter

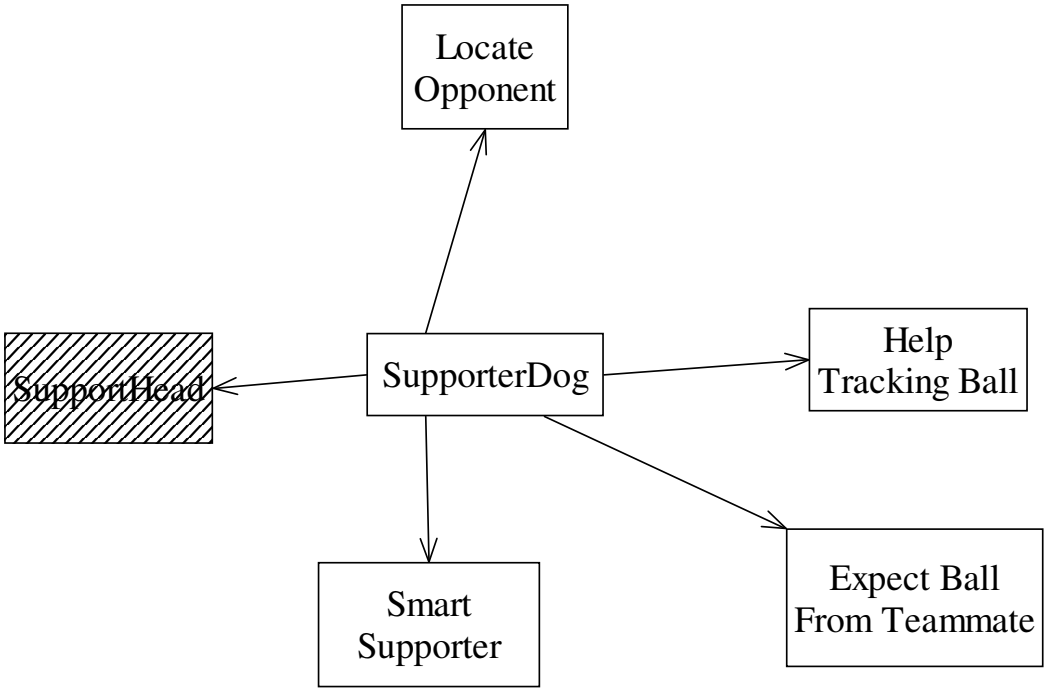

Figure 24 - Support Action Tree

The main role of supporter is to track the opponent. This is important because the attacker may face away from the opponent, thus unable to locate it. The second role main role is to position itself so that it can catch the ball passed by the attacker. These two roles are handled in SmartSupporter. The main task of SmartSupporter is to go to a particular position, and to use the head to track the opponent and the ball alternatively.

4.3. Specific skills

4.3.1. Low level skills

A number of low-level skills were made available for the challenger's behaviours. These skills are higher than SpecialAction in a sense that they are programmable. However, they are different to high level skills because they need very-little or none decision making.

4.3.1.1. Approaching Ball

This performs a slow down walking action. This is used in ball grabbing skill. In order to smooth out the transition from EllipticalWalk, which only allows the dog to control in each half step, to stop and grab the ball. In this action, the forward, and turnCCW parameters of normal walk is used. Forward parameter is calculated linearly as follows:

$$Forward = (maxForwardSpeed - minForwardSpeed) * \frac{ballD\ \ stopDistance}{slowdownDistance\ \ stopDistance}$$

maxForwardSpeed is the largest Normal walk forward, e.g : 7

minForwardSpeed is the speed at "stopping" movement. One can tune this so that the dog does not stop at stopDistance. For example, when grabbing ball, the dog can still move, but as slow as it does not hit the ball to much.

Behaviour	slow down when ballDistance<=slowdownDistance, linearly until ballDistance< stopDistance
Input	slowdownDistance, stopDistance, maxForwardSpeed , minForwardSpeed
Condition to break out of action	ballDistance > slowdownDistance or ball is lost
Condition to switch to next action	ballDistance < stopDistance

4.3.1.2. Grabbing Ball

In this action, the dog first pushes the head down without opening the mouth. It then pushes the head further with the mouth open. Finally, it checks if the ball is stuck under the chin or not.

There are two reasons the mouth is used. The first one is that the opened mouth can push the ball inside and hold it tighter. The second and more important one is that it can be used to check whether the grabbing is successful or not. Such a validation is necessary because in the game, the dog is usually pushed by other dogs; hence there is chance that it might miss the ball.

In order to validate ball under chin, two methods were used. The first one is to use the chin sensor. There is a little soft patch of about 3 cm^2 under the chin of the dog that acts as a touch sensor. Because the area of the patch is not very big, the ball does not always touch the sensor. Therefore we need another check, using the mouth sensor. The mouth sensor senses the opening angle of the mouth, in the range of 0-10e6. When the ball is under the chin, the mouth is stuck on top of the ball and can not open wider. Therefore, a simple check by comparing the angle that we told the mouth to open and the angle that the mouth is actually opening could be used to detect if the ball is successfully grabbed. The disadvantage of the second method is that it takes time for the mouth to open, while the first method does not. However the reliably of the second is almost 100%, while the first method only works when the ball touches the right spot. In fact, our behaviour combined the two methods, which has advantages of both.

This action is programmed using the time counter (see 3.3.4). For the first 5 frames, it pushes the head down. Then, it opens the mouth and pushes the head down further. After the first 18 frames, which is the time it needs to fully open the mouth. It checks if the ball is under the chin base on the two above checks.

This action can be thought of as a Special Action, which can be done in ActuatorControl. However, implementing this in Behaviour level has many advantages. It is easier that Special Action because only the head needs to be control, the leg can be controlled by PWalk parameters, which is easier than computing every joint angles. Eventhought PWalk no longer has a step complete delay; SpecialAction can not stop until it is finished. Therefore, we can not break out of SpecialAction once we detect that the ball is missed grabbed.

In addition to the two checks using mouth and chin sensors, vision is also used to detect false grabbing cases. If the ball is too far from the robot, it is impossible to grab, then grabbing fails immediately. When the dog starts opening the mouth, it should not see the ball. Therefore if it sees ball when opening mouth, grabbing fails too.

This action is in charge of only when the ball is close enough that it can use the head to grab it. The complete RunToGrabBall action will make use of this skill and will be described in the next section.

4.3.1.3. RunToGrabBall

This skill combines four lower skills: HoverToBall, GetBehindBall, ApproachingBall and GrabBall. This action makes the decision “when to grab the ball” and “what is the best way to approach the ball”. It turned out that these decisions are not trivial and influence greatly on the reliability of the grabbing action. During the course of developing these behaviours, these decisions are fine-tuned and re-tuned many times. Therefore it is essential that these behavious is encapsulated in one place, other wise each tuning involved many scattered places.

The condition to grab the ball is the ball distance is less than some distance, say 13.5 cm and the absolute local coordinate of the ball (Figure 25) is smaller than some distance, for example: 3 cm. In addition to that, because in some bad vision situation (like one in the competition). It is necessary to enforce the condition by checking it for several consecutive frames. Only when distance condition is satisfied in, say 3, consecutive frames, and the ball grabbing action is executed.

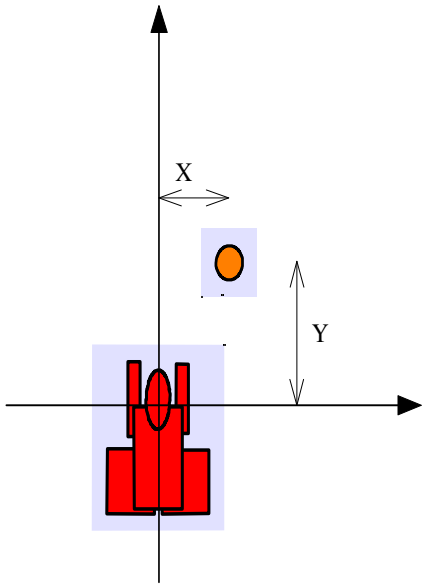

Figure 25 - Local coordinate, example: ball

Furthermore, Grab Ball action can not be done if the ball is near the edges since the paw will hit the ball. In such cases, the dog needs to get behind the ball, until it is possible to grab the ball without the paw hitting the edge (Figure 26).

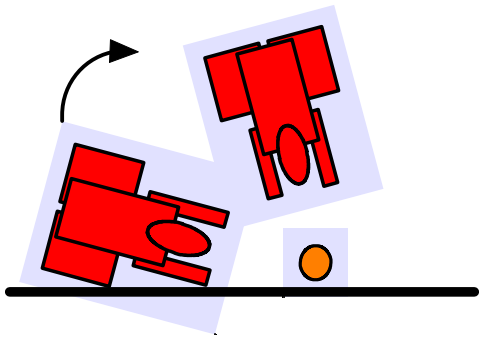

Figure 26 – Grabbing near the edge

4.3.1.4. Grab-turning

In the old ERS210 model, the dog can grab the ball, turning on its elbows, aim, and shoot the ball. However, due to a physical difference on the ERS7, the exact movements of the old skill can not be re-used. However, the new ERS7 has extremely flexible head. Similar to grab ball action, the head is used to hold the ball under the chin, and having the legs freely turn. This action is implemented in HoldTurning class in OpenChallenger.py.

There is currently a big disadvantage of using the head. In the old skill, the head does not involve in the action, hence can be used to look at the target goal to shoot, or beacons to localize. In the new skill, the head is sticked to the

ball, and can hardly see anything. The new model prevents the new dog from performing VOAK [1], which relies heavily on vision. Turning is done by using a counter, and relies on GPS before grabbing ball. The dog grabs the ball, turns for a number of frames n, and n is calculated by the simple formulae:

N = AngleToTarget * Number_of_frames_to_turn360 / 360

AngleToTarget is the difference between the desired heading and current heading, based on GPS.

Number_of_frames_to_turn360 is the number of vision frame for the dog to turn 360 degree. This constant needs recalibrating when the walk parameters are changed, or when the surface in the competition is changed significantly. In 2004 competition, we found the constant the same as the one used in our lab.

Eventhough this skill requires accurate World Model; this skill performs reasonably well in the OpenChallenger in our lab. In the game condition where there are a lot of other robots around, poor GPS prevented this skill to be useful, especially when there are robots pushing the robot.

However, it is turned out that the ERS7 can do a better Grab-Turning skill, which is demonstrated by UTS team. By changing the walking style, the head can actually both holds the ball, and look at the horizon in front of it. That type of behaviour contributed very much on UTS team and their runner-up title.

4.3.1.5. Pulling Ball out of goal

This skill is needed when the ball got hit into the goal box. Since the challenge aimed to require as little human assistance as possible, the dog should try to grab the ball out of the goal box. This skill is implemented in PullBallOffGoal class in OpenChallenger.py.

This skill composed of four actions, AppoachingBall, GrabBall, StepBackward and Grab-turning. These actions are piped one after another. The dog first approaches the ball, which is inside the goal box. When the GrabbingBall condition (see section 4.3.1.3) is sastified, it grabs the ball. It then steps backward out of the goal, and turns around using Grab-Turning skill described above to turn 180 degree from the goal.

This skill is triggered when the dog discover that it is inside the goal box. When the dog is inside the goal box, it sees almost nothing except the goal colour. Therefore detection of this situation is done by vision. Whenever there is a blob of yellow or blue goal which has the bounding box covering the whole cplane, then it is inside the goal box.

This skill however does not contribute into keeping the ball away from the other dog at all. It is implemented to avoid too much human intervention.

4.3.1.6. Turn Kicking.

Turn kick used in 2003 code is proved to be very useful. A similar skill is implemented in OpenChallenger called TurnKick. The skill consists of approaching the ball and turn-kicking the ball in either two directions: clockwise or anti-clockwise.

There are two actions implemented based on this skill. The whole approaching ball and kicking ball is implemented in DribbleTowardTeammate class. The lower skill which only is happening when the ball is under the chin until the ball is kicked away is in TurnKick class in OpenChallenger.py. The decision when to start turn-kick the ball is not in TurnKick, but is in action that makes use of TurnKick, as usuall (e.g : DribbleTowardTeammate).

In turn-kicking, the dog turns with special walk parameters, which lower and bring the two front hands forward, in order to "tap" the ball (see videos).

	Forward	Left	Turn	PG	**hF**	hB	hdF	hdB	**ffO**	fsO	bfO	bsO
NormalWalk	7	0	0	40	**90**	110	20	20	**59**	10	-50	5
TurnKick (clockwise)	3	-2	-20	40	**70**	100	10	20	**95**	15	-55	5

Table 1 - Comparison of PWalk parameters used in TurnKick and NormalWalk

The skill relies on the distance to the ball to detect when it has kicked the ball away. A threshold of 20cm is used. Whenever the distance is exceeding it, the skill will terminate so that other action can take over. When the dog missed the ball, it turns away from the ball, hence the ball distance will exceed the threshold and the action will terminate so that the dog may start the whole action again.

4.3.1.7. UPennKick behaviours.

Using UPenn kick (or side-way kick) is by far the most powerful kick in our system. One problem is getting the dog walk to the ball, and kicking it at the very right time is a bit unreliable. The reason is because hitting the ball at different spots on the paw causes the kick performing differently.

In general, we want to align the dog a bit off to the left of the ball when doing right hand kick, and to the right when doing left hand kick. Moreover, we do not want the dog to get very close to the ball, then stop, align and then kick because it is too slow. Instead, the dog should align itself when walking toward to ball. This behaviour is different from HoverToBall where the dog walks straight toward the ball in the center. It is, however, similar to paw-kick behaviour [1]. Therefore, we can reuse by adjusting the "Leg x offset" in Figure 27.

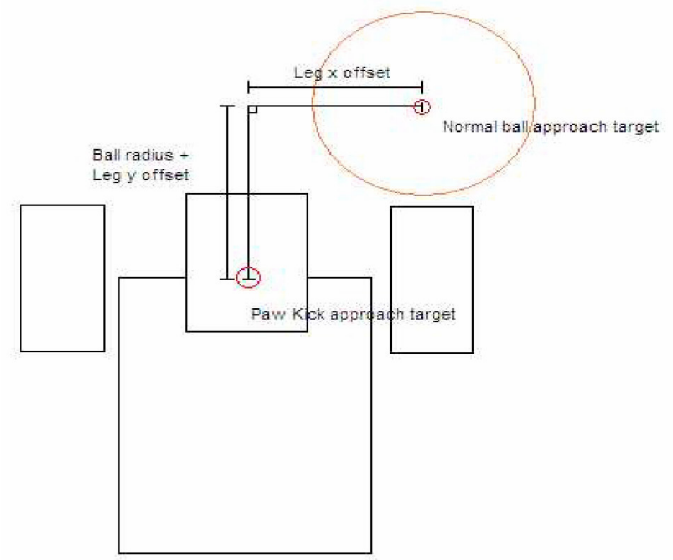

Figure 27 - Paw kick (2003 report)

Once the dog aimed at hitting the ball at the side, it only needs to wait until the right trigger is invoked to kick. The trigger condition is constraint by two values: the ball distance and "x offset" value.

The decision of executing the left hand kick is (similar for right hand kick):

Ball distance <= 14 cm:
X offset < 0: yes
X offset > 0: no
28cm > Ball distance > 14 cm:
X offset > -3: no
-3 > X offset > -7: yes
X offset < -7: no
Ball distance > 28 cm: no

This decision is illustrated by the shaded region in the figure below:

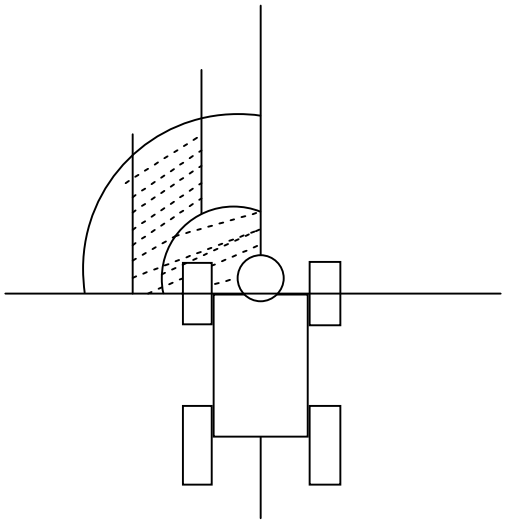

Figure 28 - UPenn kick condition

In general, this skill works pretty well. Another advantage is that this UPenn kick can be used alternatively with Paw kick because they share the same goal of walking toward the ball. Both of these are, in fact, used in KeepAway action in section 4.3.2.5.

4.3.1.8. UPennKick-ing after grab-turning

As mentioned above, perform UPenn Kick requires a lot of careful fine-tune in behaviour level. Eventhough the above behaviour guaranteed that the dog can kick the ball with the right paw, and reasonable powerful, it still does not make the kick more consistent. In our challenger, in order to successfully pass the ball to teammate, we need a more reliable way of kicking.

We realized that doing UPenn kick after grabbing the ball is much more reliable, because once the ball is grabbed; it is much less dynamic than being any where on the field. Therefore, the kick is guaranteed to hit the ball at a consistent spot.

Tuning this kick is similar to tuning turn kick addressed in 2003 report [1]. The ball is moving in synchronization with the walk step, or (1/PG). Therefore, we can find the right step value which make the kick most efficient. A trial and error experiment has been done, by having the action perform the kick at different step, and see which value perform the best. We did this experiment with UPenn kick left and right, either when it is tuning clockwise or counter clockwise. Here is the result (note that one step cycle has 2*PG steps = 80 steps).

	UPennRight	UPennLeft
Turning Clock wise	0,41,79	19, 66
Turning Counter Clock wise	18, 60	43,80

It can be seen that the good steps are at almost separated by approximately 40 steps, which is one-half cycle.

The only problem here is that the kick can only be performed at the right half-cycle, the aiming target might not be accurate, for example, if the dog wants to turn for 30 degree and shoot, because of the delay for the right half-cycle, it might over turning, say 35%...This is a trade-off between accuracy and reliability.

4.3.2. High level skills

4.3.2.1. NormalBallState: Opponent is far away, hold the ball and wait
4.3.2.2. EdgeBallState: when the ball is near the edge
4.3.2.3. PanicBallState: when the ball is in danger.
4.3.2.4. TeammateTooCloseState: block the ball and wait until teammate gets away
4.3.2.5. KeepAway: kick the ball away from opponent
4.3.2.6. SmartPassing: pass the ball to teammate
4.3.2.7. BlockingDog: try to hide the ball away from opponent.

4.4. Interaction between the 2 dogs

4.4.1. Role switching algorithms.

The role is done by a variant of token passing algorithm. A dog becomes attacker only when it has the token. Only when the attacker gives up the token, and pass to the supporter, it would then become attacker. However, the supporter may want to become an attacker. When it decided to become attacker, it first sent a request to the teammate asking for that token. The attacker then decides whether it should give up or not before actually giving up the token. When the attacker gives up the token, it immediately becomes supporter. Due to network delay, there may be exceptional case when both dogs have the same role of supporter when the token is on the way to reach the supporter. However, this moment is usually very short, and can be neglect. To ensure the token won't be lost due to network problem, it is sent multiple times by the original attacker dog. This can be done by the supporter by checking the counter whether it has just become supporter for a certain number of N frame, then it keeps sending the token N times, one per frame.

The rationale of this algorithm is that we don't want to have two attackers or two supporters at the same time. Two-attacker situation is not desired because when both of them are attacking the ball, they both lose track of the opponents, which is dangerous. Two-supporter situation is not wanted either because the ball will be left unattended.

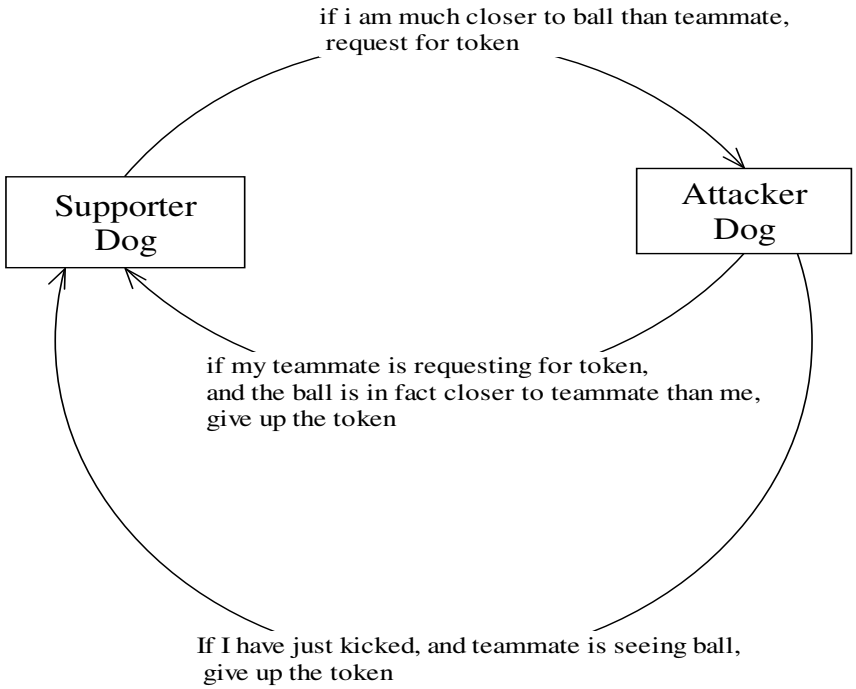

Figure 29 - Role switching decision

4.4.2. Supporter positioning

The first goal of the challenger is to support the teammate. One of the strategies of keeping away soccer is to pass as much as possible so that the opponent dog can not follow. In order to archive this, the supporter dog needs to position itself in a convenient spot for the teammate to pass the ball.

The first experiment we did on this was to have the attacker dog moving around, with the opponent chasing it. The supporter's role is to position in such a way that the ball can be easily pass to it, without any risk of losing the ball to the opponent dog. Three methods have been investigated.

- First attempt: intersect of two circles

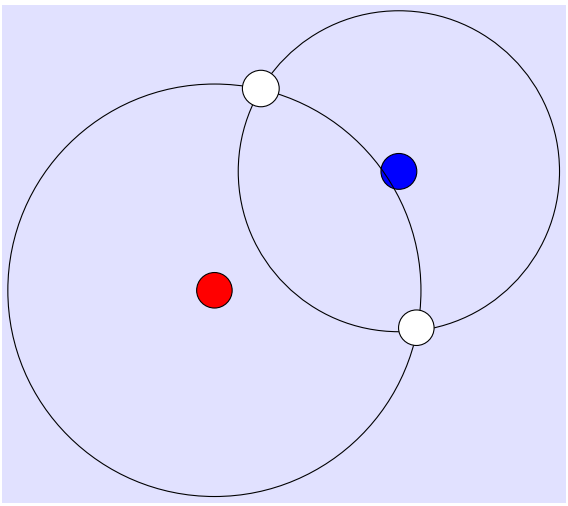

Figure 30 - Positioning 1: Two intersecting circles.

(red is teammate, blue is opponent and white is the adviced potioned)

The first and foremost condition is that the supporter must be a certain distance away from the opponent. Otherwise, a slightly inaccurate passing to the supporter with opponent being close might easily go the opponent instead of teammate. Moreover, the larger the distance, the more advantage the supporter has, to be the first to grab the ball.

Secondly, the supporter should not be too close to the teammate. This is because if the supporter dog can not catch the ball, the ball will run far away and may be caught by the opponent dog. Another issue is that we want the two dogs spread out rather than get bunched up, which is harder to control.

By the two above essential principle, it is natural to come up with a “two circle intersections” positioning as in Figure 30. Since there are usually two intersection points, the dog can simply choose the closest point to its current position. This approach is simple enough to implement, however has a few major drawbacks:

- There are cases when the two circles do not intersect each other, or intersecting points are out of the field.
- The WorldModel always has error. When the teammate and the opponent are close, imagine the two circles have the same radius. The two centers jump around making the intersecting points anywhere as illustrated in Figure 31.

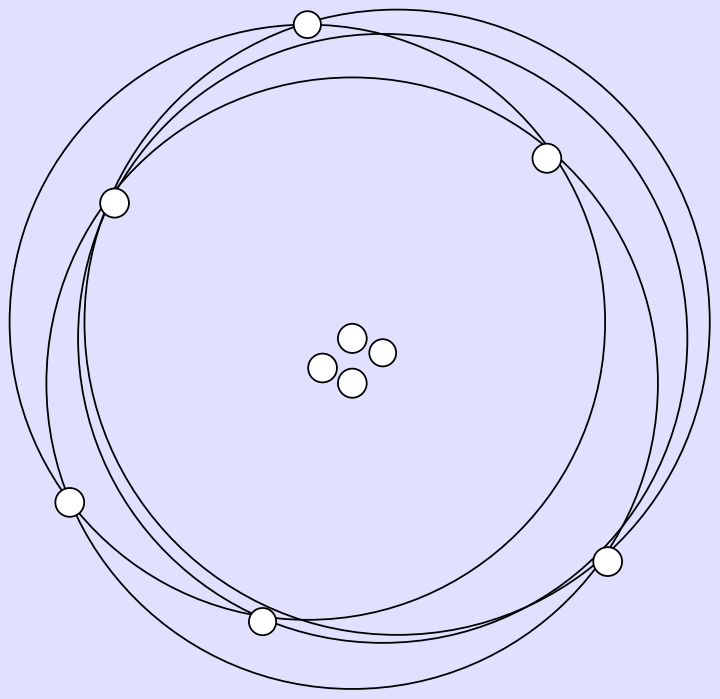

Figure 31 - Drawback: Arbitrary positions

When the two circles are not intersecting each other, they are in either one of the two cases: The centers are too close, or too far away. We just need to handle these cases separately. The first case is a dangerous case, because the teammate, who is having the ball, and is being chased by the opponent. This case is handled in Figure 33. The second case is when the opponent is too far away. In this case, the teammate should not pass the ball immediately, but instead, hold the ball and wait until the opponent comes closer. The supporter can do "Marking" as in Figure 32.

Figure 32 - Positioning 2: Marking

The supporter can mark the opponent by getting in the way so that the opponent can not directly see the ball; hence maximize the ball keeping away time. This case can be modified so that the two dogs can do passing yet hiding the ball away from the opponent. However, this behaviour has not been achieved yet.

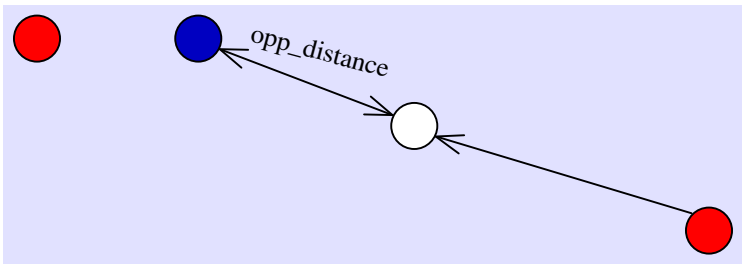

Figure 33 - Positioning 3: Teammate in danger

When teammate is in danger, it is difficult to locate exactly which direction to support. The supporter just hangs back keep the distance with the opponent. It is then up to the teammate to kick the ball away in any direction and chances are that it will be away from the opponent.

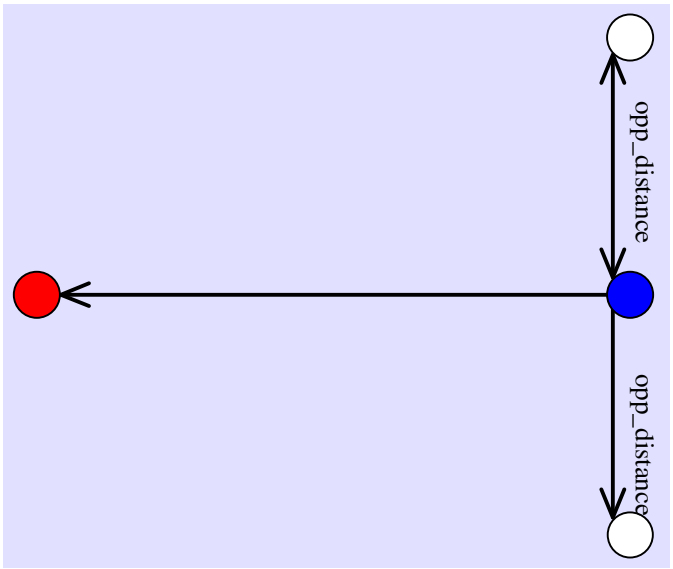

Figure 34 - Positioning 4: Right triangle

Another attempt is Right Triangular positioning illustrated in Figure 34. This is somewhat similar to the two circle positioning. There are two main advantages of this method. The first one is that it is easy to compute. The second one is because the angle of teammate and opponent to the support is always less than 90 degree, it enable the supporter to visually track the opponent as well as teammate and ball.

A special case when the opponent is in the corner, it is better to position into one of the four pre-defined position as below:

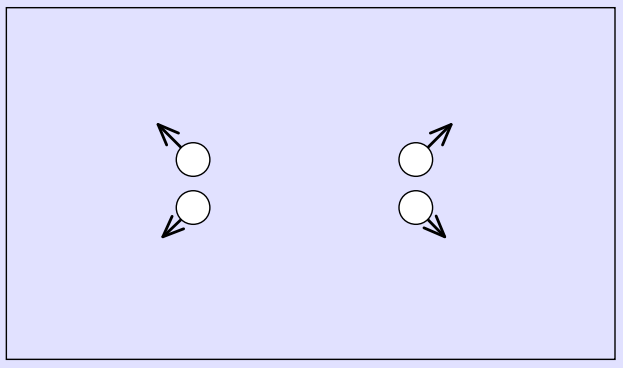

Figure 35 - Opponent in corner case

The rationale is that those are the best position to track the opponent and still avoid moving. If it is necessary, the supporter can switch to attacker, and go away to find the ball.

In the end, all of these strategies are used except two-circle intersection. Square-triangle positioning is used as the general case, other strategies handle special cases (opponent in corner, teammate in danger, and marking opponent).

4.4.3. ExpectBallFromTeammate

The attacker dog does passing by first sends a message telling the supporter dog to expect it. The supporter is then in a state of "pre-attacking" ball. It locates the ball, based on the assumption that the other dog has just kicked the ball away.

This "pre-attacking" action is called ExpectBallFromTeammate in OpenChallenger.py. Whenever the supporter receives the message from the attacker dog, it knows where the other's facing. Guessing the other dog would kick the ball straight ahead, and assuming a fixed distance D_KICK of the kick (100cm), the supporter would run toward the expected ball position, rotating the head to find ball at the same time. However, after small amount of time (2 seconds), if it doesn't see the ball, it should give up the assumption that the ball is D_KICK distance away from the teammate (the world model of the teammate may be wrong). When the supporter gives up, it turns back to locating ball action.

4.4.4. Collaborative tracking ball.

At the same time as the supporter dog doing ExpectBallFromTeammate, the attacker dog is becoming SupporterDog. It detects if its teammate has found the ball or not. If not, it continues tracking, sothat the teammate knows about wireless ball. This action is done in HelpTrackingBall, and is executed in SupporterDog, whenever its teammate loses the ball.

This behaviour only works when the wireless is nearly perfect, which is not very true in the competition.

4.5. Conclusion

The OpenChallenger somewhat works almost as well as planned. However, there were a number of problem that prevented it to perform well in the competition.

The first problem was the changes in environment. Vision and Locomotion were affected significantly. The rough carper in Lisbon prevents our one-handed kick to function properly. In the lab, the ball is usually kicked away about 180-250 cm, while in Lisbon, it reduced to roughly 100-150cm. Unfortunately, we didn't have many alternative kicks to overcome the problem. The lightning condition is also different to the one in our lab affects sanity checks so much.

The second one was network problem in Lisbon. Due to the poor wireless network at the competition, the communication of the two robots was affected seriously. A large part of the challenger heavily relied on the two dog communication. For example, the attacker relied on its partner to localize opponent and send that information over the network. Therefore, without a consistent network connection, the dogs faced a big problem. Furthermore, without the network, the team could not send many CPlane, therefore could not test vision offline efficiently. Sanity checks were not well verified in the new field consequently.

However, The OpenChallenge still shows a good example of applying the new behaviors frame-work. Even though it is developed in only three weeks, it has explored many of new boundaries for a behaviors challenge.

In the future, we hope that we can solve such challenges more efficient.

5. Python development framework

5.1. Introduction

Using a scripting language for programming robot behaviours has shown many advantages. In the past, University of Pennsylvania has had a big success by using their Perl language as a high level programming language for the dog [14]. The team the first time became a runner-up. At the same time one of our team member, Nicodemos has experimented using Prolog as a high level language, based on GoLog[15]. However, the works was still in experimental stage that was unable to be recognized further.

The original goal of embedding Python was to develop the Open Challenge, which is an extensive behaviour challenge. Without such a tool, it is very hard to make the challenge done in time. This Python frame work can be seen as the back-bone of the challenge.

Python is not only a scripting language, but also an object oriented language that has been used widely for both industrial and research purposes. The great advantage of Python is rapid development methodology. Since it is a scripting language, the code does not need to compile. Therefore coding and testing can be done at the same time, which is one key characteristic of rapid development methodology.

The need a rapid development method is getting more and more apparent. After Australian Open, the inadequate of the old C++ behaviours were well recognized. The team had only two months until the world competition. The fact that the current C++ behaviour code took well over 4 months in the past made us decide to use Python not only for developing the challenger, but also the game players' behaviours.

5.2. The Background

Process	Time
Compile code (minor change in Behaviour)	20s
Compile code from scratch	4m10s
Install to MS	22s
Boot the dog	26s
Total time	6 minutes 18 seconds

The current development cycle takes total over 6 minutes for a change to be made into the robot's behaviour, not including testing. This makes it undesirable to make many changes to the code. One solution is to compile any test the code offline, without any touch to the robot. rUNSWift 2003 had put a lot of effort to make this happened (page 58, 2003 Report).

However, to reproduce the error if anything goes wrong; all the CPLANEs need to be logged. This work pretty well with Vision and GPS module, because bugs in Vision and GPS do not frequently happen. There is no tuning involved in those modules. On the other hand, writing behaviours needs a large amount of tuning. The team rUNSWift 2003 has spent a huge amount of effort to write and fine-tuning behaviours, which is reported in 113 out of 264 pages in their report.

Another solution is to have a complete simulation of the soccer environment. This approach is attempted by several teams including the 2004 champion. This approach allows every phrases of the development to be done offline, which turns robot development close to traditional software development. However, the solution is only achieved to a certain level of reality. [9]

	Lines of code
Vision	12462
GPS	4542
Behaviors	27822
Wireless	1963
ActuatorControl	10320

Table 2 Comparision of amount of code in each module

A more feasible approach is to change the way behaviours is developed. As a matter of fact, compilation, install to dog and rebooting processes can be

eliminated. Instead of writing Behaviour in C++ code, which needs re-compiling, programming Behaviour in an interpreting language can remove all compilation hassle. That is the reason why Python is used in the our system.

	Without Python	With Python
Make	20-60s	n/a
Install to MS	22-25s	n/a
Boot the dog	26-30s	n/a
Upload to dog	n/a	2-5s
Reload module	n/a	5-10s
Fix-a-bug total time	68-115s	7-15s

The old way of writing behaviour, a cycle of 1)writing code, 2)make the code, 3) install to memstick, 4) boot the dog takes about 68-115 seconds, while Python simplified the process to 1)writing code, 2)uploading code, 3)reloading code in a total of only 7-15 seconds. This is clearly a win of using Python.

5.3. The development cycle

With python, the development cycle is done in the following four steps:

1. Programming in Python

Python source code can be written in any editors, however, it is the most convenient to be written in an integrated development environment (IDE) for python. The IDE we used was Eric3 [16]. It provides a wide range of tool, from debugging to code formatting, auto completion…

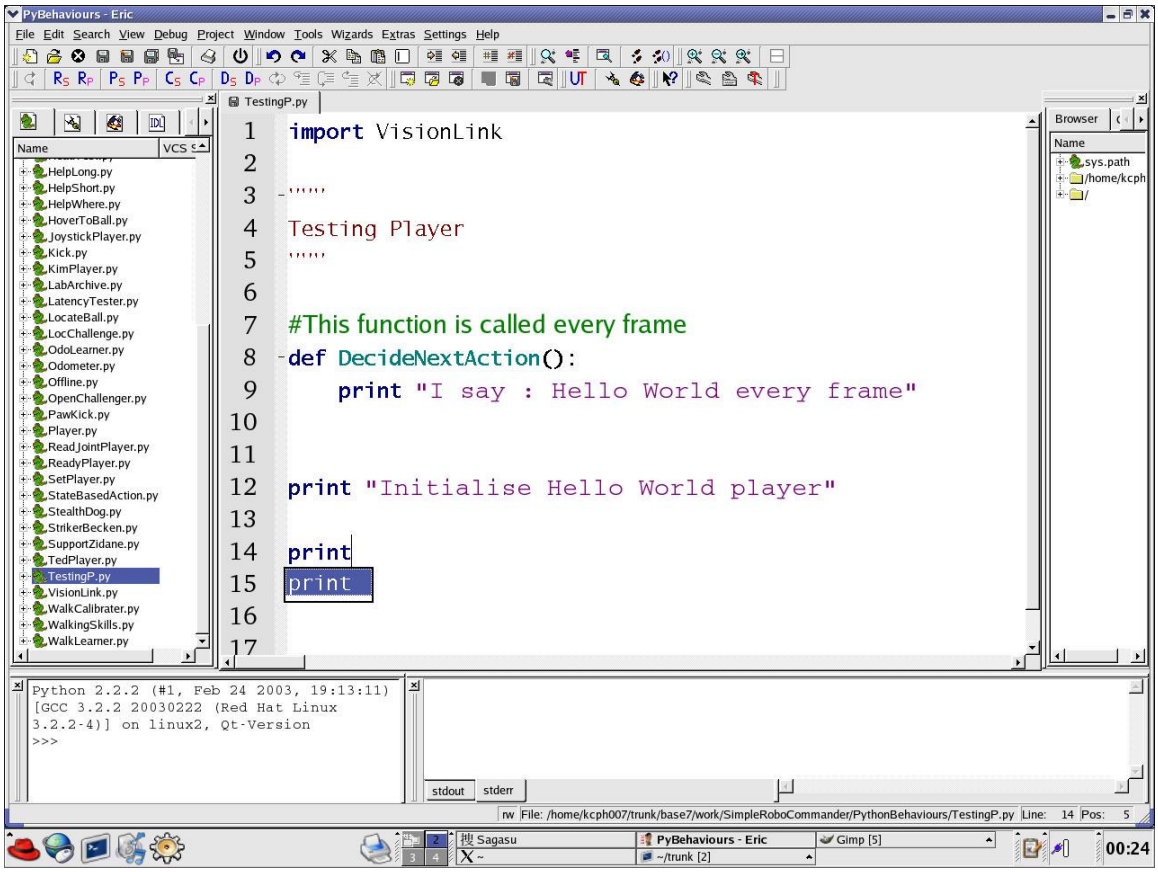

Figure 36 - ERIC3 IDE with HelloWorld Behaviour

2. Upload python source code to the dog.

An uploading tool is integrated into base station, allowing us to upload the source code in a convenient and efficient way.

The tool first finds which files have been modified.

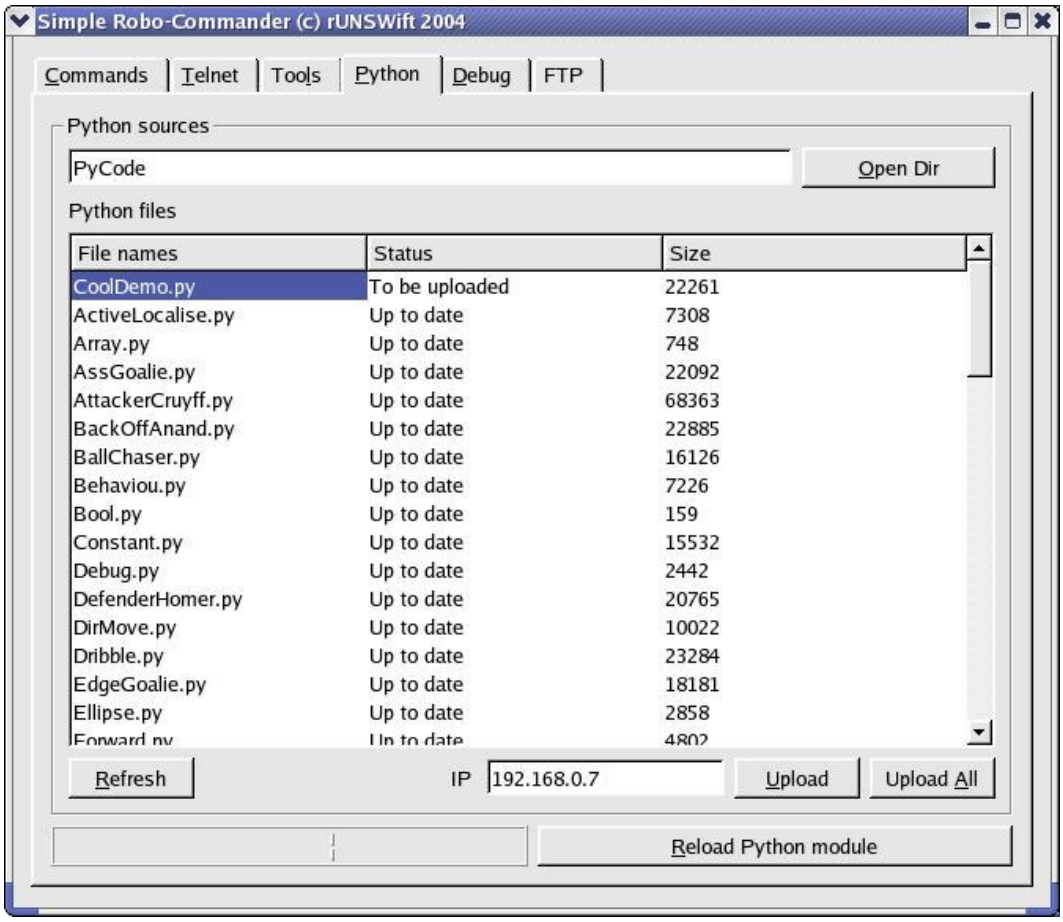

Figure 37 – Python source uploading tool

It is noticed that the python interpreter installed on the dog does not checks the existence of newer source file version. Therefore even the source file is uploaded correctly; it does not reload python modules on-the-fly. Therefore we need to tell the main module to reload by listing all the modules that needs reloading in a file called Reload.py. However, it is tedious to do it every time we change a source file. The uploading, in fact, does this for us. Note that new source file should be always reloaded properly except the dependency issues (see section 5.8)

The uploading tool also does another tedious task, checking python code by pychecker. In Python programming, it is very common to make error in runtime because source code is not compiled. On a PC this is not a problem because running a program is just as fast as compiling it. However, on an embedded system like our AIBO dogs, it takes time to upload the source, initialize modules… before it can be run. Therefore, it is the best if the program can be "test-run" on a PC which is much faster than the dog's CPU. Fortunately, there is a tool that does the "test-run" called pychecker. With pychecker, most error is picked up, such as syntax error, mistyping variable name … It also gives

warning, which is very likely to be an error. The uploading tool is also the most suitable for this task.

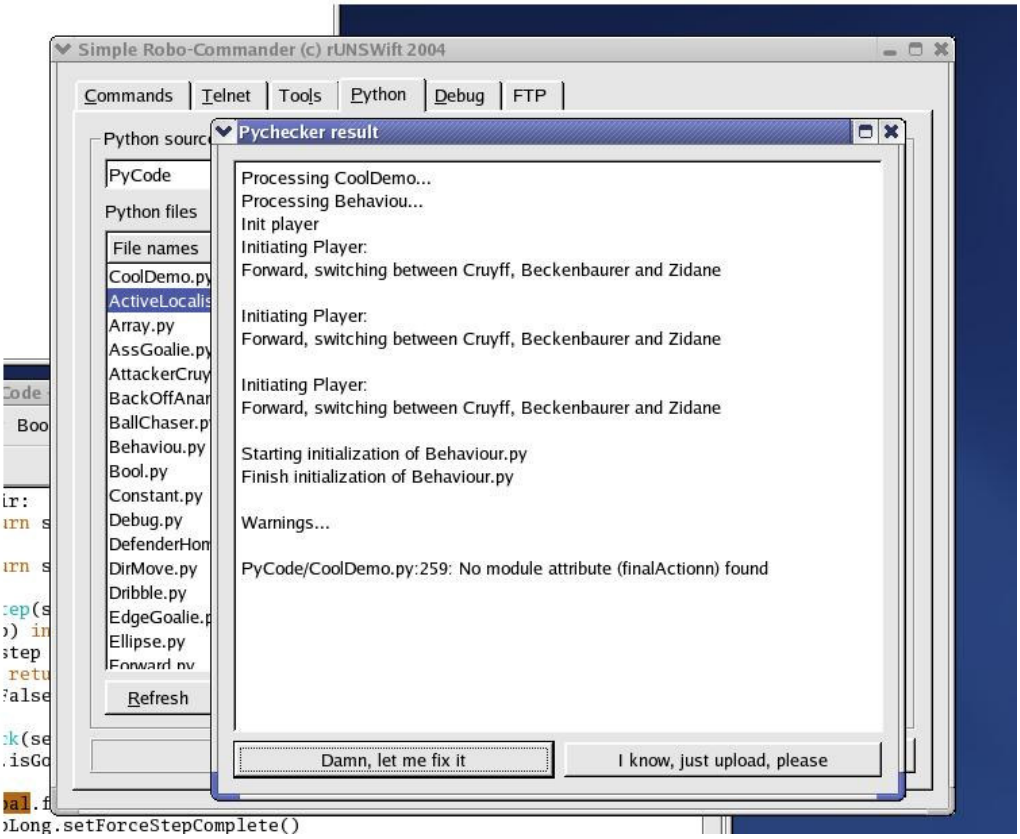

Figure 38 - Running pychecker

The pychecker warns that a variable name “finalActionn” is not found. It is clear that “finalActionn” has been mistyped here.

3. Reloading python program on the dog.

Once source code is uploaded, it is time to tell the running interpreter on the dog to reload its compiled python program. A command is sent from the base station to C++ behaviour module to reload python.

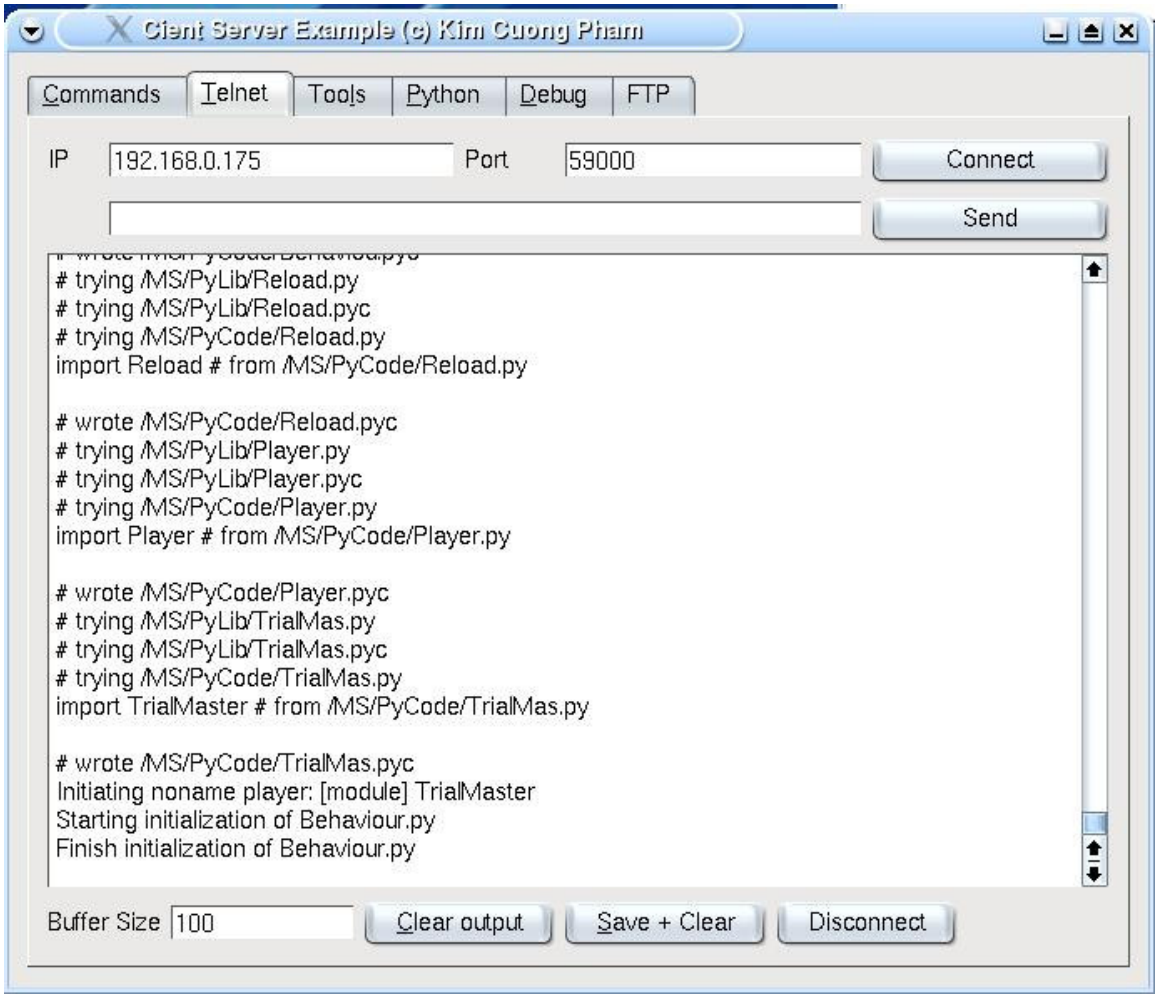

Figure 39 - Dog reloading python code

4. **Testing program.**

After reloading, the dog is ready to run the new code. The dog can now start and run the new code. If it is paused before, it can be resumed (see 5.6.1). The most common way to testing the code is to have the dog behave in various situations and watch it carefully.

This is the final step in the development cycle. After testing, if the code needs fixing then we go back to step 1 and continue the loop.

5.4. Extend the current system structure

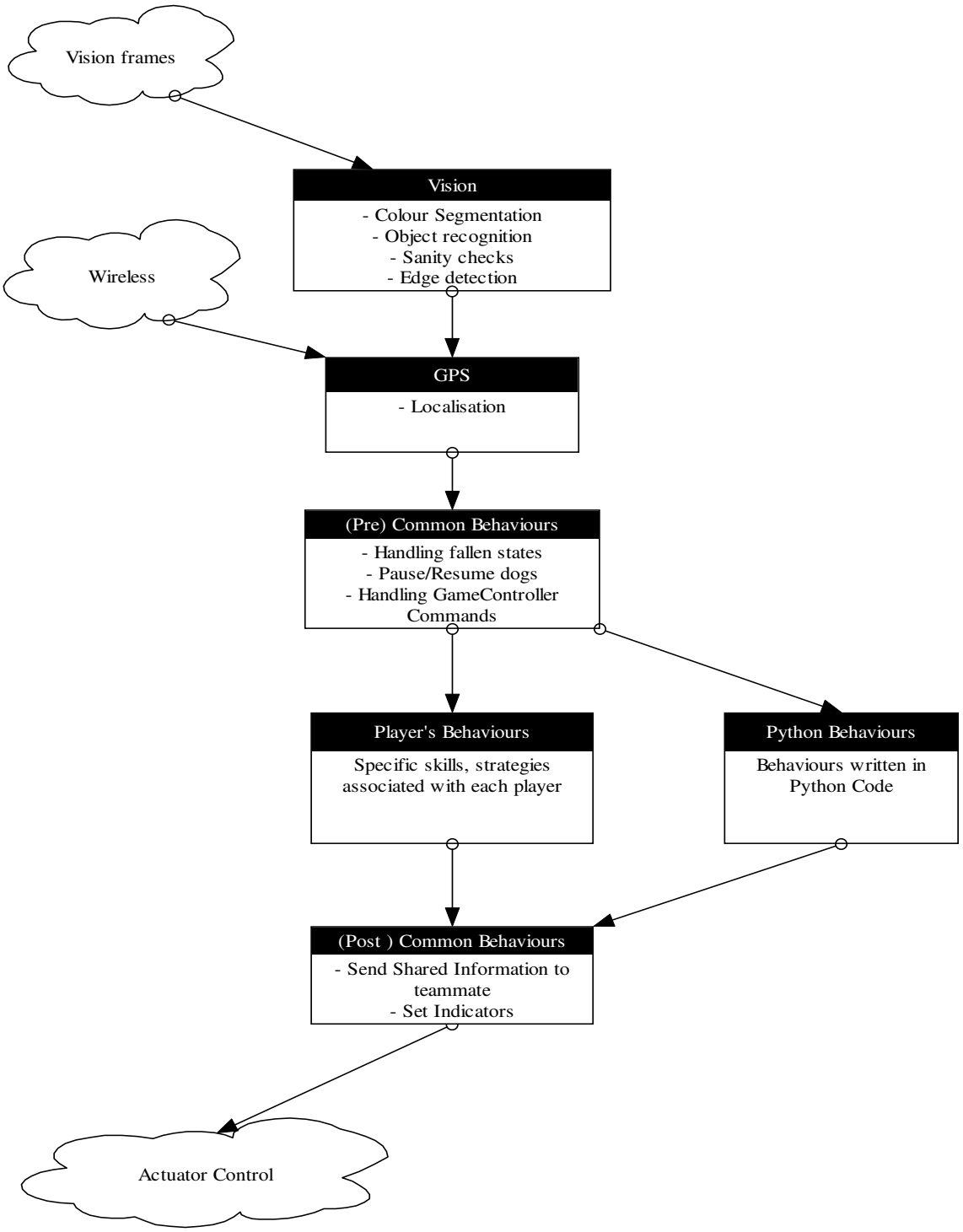

Figure 40 - Flow chart of robocup modules

The current rUNSWift robot architecture contains 5 modules: Vision, GPS, Behaviours, ActuatorControl and Wireless [2003 report]. The flow of these modules is depicted in Figure 40. Vision module is fed in vision frames by OPENR. It does colour segmentation

and forming blobs. It then recognize object and the output goes to GPS. GPS, in turn, take objects information as well as wireless information from Wireless module, to create a world model. Behaviour module takes all available information from previous processes, to make decision and decide which action to do next. The action commands are performed by ActutatorControl, where it closes the execution loop.

The python module aims at reducing behaviors development time.

5.5. Porting to Python

5.5.1. C++ to Python interface

C++ to Python interface is the C++ API functions that can be called from Python. Python interpreter provides an interface for embedding C++ function called PyMethodDef.

PyMethodDef has 4 fields [17]:

Field	C Type	Meaning	Example
`Ml_name`	char *	name of the method	sendAtomicAction
`Ml_meth`	PyCFunction	pointer to the C implementation	VisionLink_sendAtomicAction
`Ml_flags`	Int	flag bits indicating how the call should be constructed	METH_VARARGS
`Ml_doc`	char *	points to the contents of the docstring	"Set the atomic action to be executed in ActuatorControl"

Table 3 - Python Method struct

Ml_flags can have a number of flags. However, only 2 flags are most commonly used: METH_VARARGS if the function has arguments and METH_NOARGS otherwise.

The arguments and return values of these methods can be anything. It can be function pointers (which will be utilized in the next section), primitive types like integers, char, or objects… Passing Objects will be described in section 5.5.3.

Making the C++-to-Python interface is as simple as making a list of PyMethodDef struct.

```
/* With this function specifying all the implemented semi-Python/semi-C
** functions, the python codes can call the "bridging" functions implemented in
** this file.
 *
 * NOTE: CHANGE IN THIS TABLE SHOULD BE TYPED IN VisionLink.py as well!
 **/
static PyMethodDef Vision_methods[] = {

   /*
    * General functions
    */
   {"setCallbacks", VisionLink_setCallbacks, METH_VARARGS, "Set the PyEmbed callbacks."},

   /*
    * Actuator related functions
    */
   {"sendAtomicAction", VisionLink_sendAtomicAction, METH_VARARGS, "Set the atomic action."},
   {"sendIndicators", VisionLink_sendIndicators, METH_VARARGS, "Set the LEDs, tail and ears."},
   {"sendMessageToBase", VisionLink_sendMessageToBase, METH_VARARGS, "Send wireless message
back to base station, message is prefixed by '****'"},
....
```

Figure 41 - Listing of API functions (extracted from PyEmbed.cc)

Since Python is an object-oriented language, these methods must be put inside a module (Module is a pretty much equivalent to object, except that it is singleton). For simplicity, we put all methods into one module called VisionLink. Perhaps, another better way is to put them in several modules, such as Vision, GPS, Actuator…However, in term of efficiency, the two methods are the same.

5.5.2. Python to C++ interface

In order to use python code, it must be called from C++ code. In the previous section, we know that Python can give arguments to a C++ function. That is used to pass C++ the python function pointer, or callbacks. Having got the python callbacks, C++ code can call them any time, or send messages to Python modules...

We provided 2 Python function to be used in C++: processFrame and processCommand.

- processFrame is to be called every time a vision frame is received. More precisely, each time a vision frame is received, vision and gps is processed, and then processFrame is called.
- processCommand is called to pass the message that wireless module is just received to Python.

A sequence diagram will illustrate the use of processFrame and processCommand in details

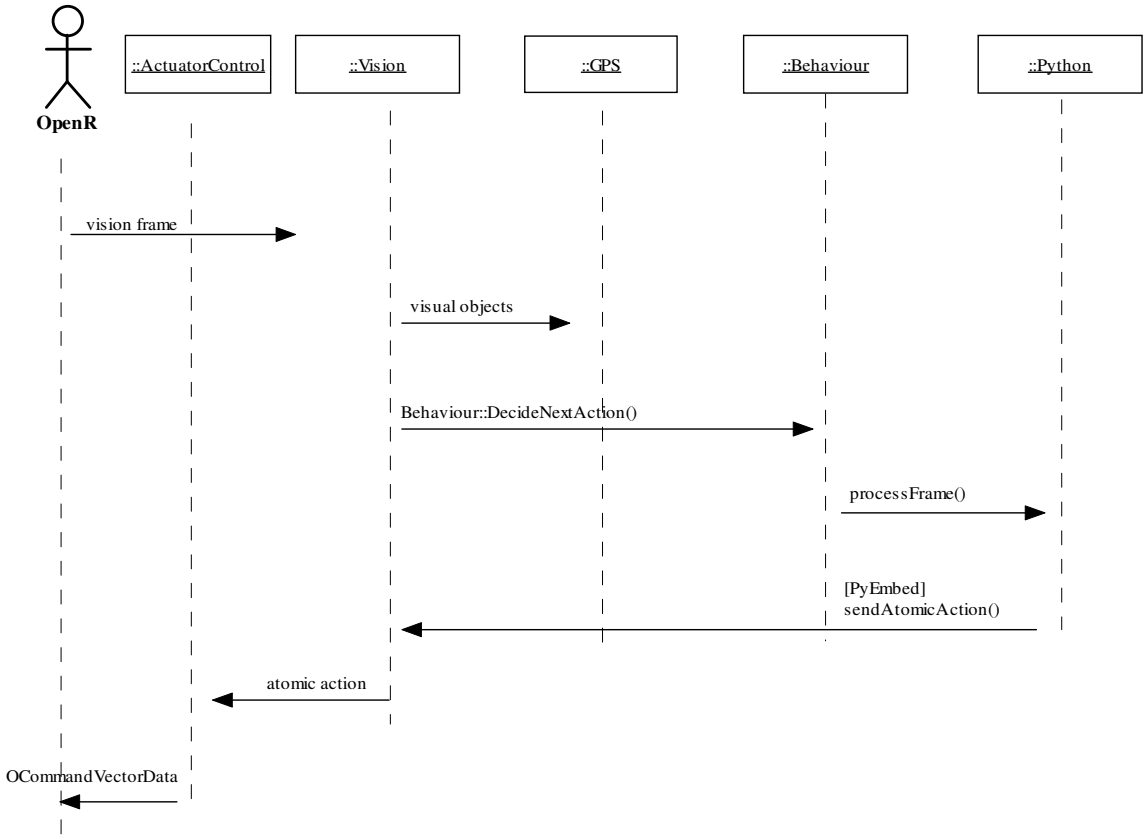

Figure 42 - System interaction with Python module

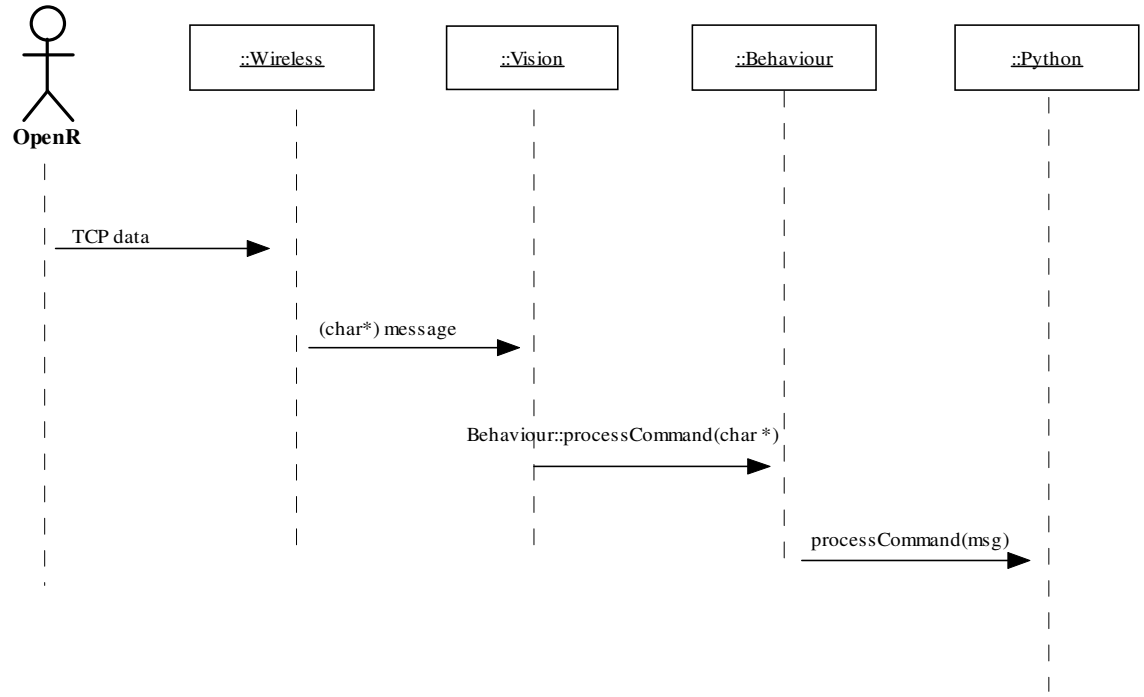

Figure 43 - Wireless message passing between modules

5.5.3. Pseudo Object

As mentioned in the previous sections, it would be convenient to pass objects as arguments rather than using numbers. However, making new object as a native python object is quite complicated and takes time. There is too much house-keeping function to be implemented [18]. We used a simpler method, which is easier to implement.

Each C++ class is serialized into a tuple, and passed onto Python from C++. We created a python wrapper object that is responsible to convert the tuple back into a python object. The wrapper object is called pseudo object, because underlying it is not a true data structure but a tuple. To python level code, the pseudo-object looks the same. It has the same methods as in C++. The only different is that the object needs to be constructed by giving it the tuple received from C++. To demonstrate this idea, let's look at Vector class:

In function getGPSOpponentInfo (in PyEmbed.cc), it returns:

```
return PyObject_FromVector(gps->getOppPos(opponentNumber, context)) ;
```

PyObject_FromVector is a function to convert Vector object into a 2-tuple containing x and y coordinate.

```
static PyObject * PyObject_FromVector(const Vector &vector){

  PyObject *t;
```

```
    t = PyTuple_New(6); //create a tuple of 6 elements
    PyTuple_SetItem(t, 0, PyFloat_FromDouble(vector.x) );      //set each element to be a
    PyTuple_SetItem(t, 1, PyFloat_FromDouble(vector.y) );      // field from Vector object
    PyTuple_SetItem(t, 2, PyFloat_FromDouble(vector.d) );
    PyTuple_SetItem(t, 3, PyFloat_FromDouble(vector.theta));
    PyTuple_SetItem(t, 4, PyFloat_FromDouble(vector.head));
    PyTuple_SetItem(t, 5, PyFloat_FromDouble(vector.angleNorm));

    return t;
}
```

The tuple is converted back to Vector object in Python by Vector class (extracted from Global.py) :

```
class Vector:
    def __init__(self,*arg):
        if len(arg) == 6:
            self.__x , self.__y, self.__d , self.__theta , self.__head , self.__angleNorm = arg
        else:
            print "arg length = ", len(arg)
            raise Exception("Number of argument to Vector() must be 6")
    def getX(self):
        return (self.__x)
…
```

Whenever the getGPSOpponentInfo gets called in Python, it is wrapped with the pseudo-object (extracted from HelpLong.py) :

```
gpsOppPos = Global.Vector(*VisionLink.getGPSOpponentInfo(i, coordinate))
```

From there on, gpsOppPos is used as a native Python object.

Five C++ class is ported this way, which are Vector, WMObj and VisualObject, PWalkInfo and WirelessTeammateInfo.

5.5.4. Global.py

The original purpose of this file is to contain global variables, which should be restricted in OO Programming. However, the existence of some non-standard API made this module become popular. “Global.py” is the place to standardize and store wrapper class for pseudo-objects mentioned in previous section. During the course of development, we realized that people preferred to access information from variables (e.g: Global.selfLoc) rather than from function (e.g: VisionLink.getGPSSelfInfo()). Therefore, such variables were put into “Global.py”. This explained why “Global.py” is almost always imported in every modules. However, in the near future, this should be changed to its original purpose.

5.5.5. Summary of the API

This is the list of function available in Python module at the time of this writing. For an up-to-date list, along with explanations of usage, please consult VisionLink.py or online documentation [19].

Locomotion function	Vision functions	GPS functions	Wireless functions	Helper functions
sendAtomicAction sendIndicators getBatteryLevel getFallenState getPressSensorCount getJointSensor getAnySensor getJointPWMDuty getPWalkInfo getBatteryCurrent getTemperature getCPUUsage setWalkLearningParameters	getVisualObject getColouredPixelCount setUseBeacons getProjectedBall getHeadingToBestGap	setGPSGoal setIfGPSMotionUpdate setIfGPSVisionUpdate setAllowPinkMapping getUsePinkUpdate getIfGPSVisionUpdate setGPSPaused getGPSSelfInfo getGPSSelfCovariance getGPSBallInfo getGPSBallVInfo getGPSBallMaxVar getGPSVBallMaxVar getGPSOpponentInfo getGPSOpponentCovMax getGPSTeammateInfo getOppGoalInfo getOwnGoalInfo getGPSCoordArray getTeamColor getGPSTeammateBallInfo setMinimalMode	sendMessageToBase sendEnvironmentTeamMates sendCompressedCPlane sendOPlane sendEnvironmentBaseStation sendYUVPlane sendMyBehaviourInfo getSharedBallInfo getWirelessTeammateInfo getTheCurrentMode getKickOffState	setCallbacks doBasicBehaviour getMyPlayerNum getCurrentTime getProjectedPoint PointToHeading PointToElevation getTestingInfo

Table 4 - Summary of Python API

5.6. Debugging/Remote Debugging

5.6.1. Four levels of continuation.

When programming on a PC, debuggers such as gdb is able to stop the program, so that programmer can examine various aspects of the running code such as stack trace, memory variables…

On the AIBO dogs, there have never existed such tools. Therefore, programming in C++ for the dog is much harder than for PC. However, with Python, it is simply possible.

Another advantage of pausing the dog is that it lets us reproduce the error. Usually, when the game is kept playing, a bug can be seen, but it is very hard to debug because it is not reproducible, unless the game is played again.

There are four ways in which the dog can be stop for programmers to examine the state of the dog:

a. Pausing the leg only: “The dog stands still while thinking…”

This is done by deleting all atomic action except head control after DecideNextAction(). This enables us to examine the code when the dog is still executing the code. However the dog will not move makes it much easier to reproduce the error. The head is movable so that localization is remained the same.

This type of discontinuation is useful when debugging high-level strategies, role determination…

The base station command[*] is "pyc/pleg n", where n is the dog number[†].

b. Pausing both leg and head: "Watch closely and think carefully…"

This stop the dog both legs and head. This makes the CPlane that the dog is seeing stay still. It is particularly useful when a bug is caused by vision.

The base station command is "pyc/n pause" where n is the dog number.

c. Stopping DecideNextAction(): "No Python."

DecideNextAction() is not going to be called, therefore the state of the player's behaviour code remains the same. General python behaviour like processing wireless command is still enabled.

This is often used when it is not clear which proportion of the player's behaviour has bugs.

The base station command is "pyc/n nodna" where n is the dog number.

d. Stop everything (mode 0): "No Behaviours."

This is the old pause command that stops the dog right from the C++ to cease calling DecideNextAction() [‡]. The dog cannot communicate at all, except for sending cplane and world model, until it is started again in mode 1.

The base station command is "mode/0".

5.6.2. Examining variables by dynamic evaluation

Having stopped the dog and the code it executes in the previous section. We now look at how the state of the program is examined.

In Python, it is possible to evaluate any expression at run time. As a matter of fact, Python interprets and executes everything at run time.

To evaluate an expression, we send the expression in a wireless message to the dog, the message is then handed to Python, it extracts the expression, evaluate

[*] Base station command has 2 fields: name and value. By convention name is to demultiplex in C++ level, field value is sent to specific player's code. E.g : name=pyc means message for Python player.

[†] Dog number can be 1...4 or 5...8. "9" means all dogs. "0" means no dogs.

[‡] C++ DecideNextAction() is different from Python DecideNextAction(). The former calls the later.

it, and send back the result via a Python embedded function called VisionLink.sendMessageToBase(msg).

Note needs to be taken when evaluating the expression. The expression is evaluated in Behaviour.py, which is the top level module in Python. Therefore, it is interpreted in the context of Behavior.py module. In order to refer to a variable in a specific player's code. It needs to be accessed though module name, imported from Behaviour.py. For example, suppose there is a variable "frameCounter" in the player's module (say, Testing). The expression has to be "Player.player.frameCounter". Because "Player" is a general player module imported by Behaviour. Testing player is in turn imported as "player" in Player module. Therefore, the expression has to refer to all these modules.

(A gotcha is that since a symbol needs to be accessed via its modules, all symbols that are imported by "from ModuleX import *" will not be accessible via "ModuleX" but accessible via the module that does import.)

5.6.3. Enable debugging by dynamic execution

Dynamic evaluation only read the state out of the program, while dynamic execution can actively change the state of it. Again, this is another feature of Python. The use of execution is mostly similar to evaluation.

This can be used to change the value variables to simulate from erroneous condition. The most often use of this is to turn on/off some debugging variables, which is seen from SanityCheck player (see Appendix 2). The player is used for either sending CPLANE that visual objects (beacon, ball…) are not recognized (false positive) or contains visual objects that is not exist (true negative). By changing one of 9 variables (one for each type of object), we can get a lot of bad CPLANE for debugging in offline vision.

5.7. Communication with base station

Communication between the dog and base station is never been as easy as it is with Python. Almost all Python players take advantages of this communication, especially responding to base commands. WalkingLearner makes the most out of this communication by both sending and receiving commands. A number of debugging, testing, experimenting player make use of wireless communication: ReadJointPlayer, LatencyTester, LandmarkTester, Odometer, OdoLearner…

5.7.1. Listening to commands from base station

Listening to commands from base station is handled by processCommand function in Behaviour.py. A simple example extracted from Behaviour.py is given:

```
#-------------------------------------
# Called when received a wireless command.
def processCommand(cmd):
        print "Python:I just got this command: ",cmd
```

Recall that any message from base station to the dog is in the form of name/value. The name field is processed by C++ code, usually to forward to a particular C++ player. The name of messages to Python player is "pyc" (Python commands). The value field is formatted as "n message", where n is the dog player number. This number is also processed by C++ layer. The rest of the message will be given as an argument in the above processCommand.

5.7.2. Sending data to base station

```
def processCommand(cmd):
        …
        if cmd[:4] == "eval":
                evalResult = eval(cmd[4:])
                VisionLink.sendMessageToBase("****"+str(evalResult))
        …
```

Sending message is done via function sendMessageToBase in VisionLink module. The above code is sending the result of an expression evaluation to the base (see 5.6.2). The message is first passed from Vision to Wireless module via the share memory. The share memory structure is defined in SharedMemoryDef.h: RC_DEBUG_DATA. The struct field "message" is used to pass the data. However, that "message" field is limited to 100 characters. Therefore the current code only allows maximum of 100 characters to be sent from the dog at a frame. This explains why some message from the dog is truncated to 100 first characters. This will be probably fixed in the next version.

5.8. Other issues

There are some miscellaneous issues involving in Python.

- Dependency issues:

  When a Python module is reloaded, its imported modules are not automatically reloaded. This makes sense because if it is, then circular dependent modules will take forever to reload. Therefore, we had to force it to reload a special way.

  The way we did it was to have a module called Reload. It is forced to reload by a statement "reload(Reload)." in Behaviour.py. Therefore, everytime Behaviour is reloaded. It will reload Reload as well. Reload module basically contains all the modules that need reloading. The modules that need reloading are usually modules that are modified, which can be generated by Uploading Tools (see 5.3).

  There is a problem with reloading imported objects from a module by "from moduleB import *". When using this statement, reloading moduleB will not reload all contructs that has been imported by moduleA. Therefore, in order for a module to be reloaded properly, this statement should be avoided.

- Printing out to telnet:

  Printing out from Python may slow down the dog due to threading issue. This is because the interpreter running on the dog is single threaded. All the I/O processing is busy-waiting. Moreover it is sent over wireless and the delay large. Therefore, printing a large amount of data to telnet should be avoided.
- Testing Action:

  Any single module can be run as long as it provides DecideNextAction(). This is similar to Java static main function. It is useful when testing a module. The module can be treated a single player.
- Player's description:

  When Python module is reloaded, the player's name and description are displayed. This is to avoid loading wrong modules. For convenient, the player's name and description are put in a triple quotes:

```
"""
Forward, switching between Cruyff, Beckenbaurer and Zidane
"""

import VisionLink
import Debug
…
```

The string inside the triple quotes is interpreted by Python as a special description belongs to the modules. This special string is printed in Player.py

```
…
# Print player description everytime it is reloaded
if player.__doc__ is not None:
   print "Initiating Player:", player.__doc__
else:
   print "Initiating noname player: [module]", player.__name__
…
```

5.9. Future improvements

There is a lot of room for improvement of the Python development framework. The API can be made more Object Oriented by grouping them into appropriate module, such as Vision, GPS, Actuator… The pseudo-objects (5.5.3) could be replaced by proper embedded objects for better performance. The accessing of information from C++ should be refactored to be more OO, instead of accessing global variables in Global.py. Wireless communication could be made more standardized or extended to send more type of data rather than raw character messages. Another importance development aspect of the framework is the

SimpleRoboCommander base station. The base station could be improved to be more user-friendly, more fault tolerance. All these things can definitely boost our development efficiency and quality.

6. Supplementary Tools

Tool is one of the key aspects of success in our project. Along the course of developing the system, there are a number of tools that supplement our developments. There are two tools that I mainly developed: CPlaneClient in C++ and JointDebugger.

6.1. CPlaneClient

This tool is originally developed in the early of the year (Summer 2004), to experiment with the throughput of sending CPlane. The full frame rate of the dog's camera is 30 frames/sec. However, the old Java CPlaneDisplay (see report 2003), can not display CPlane at full frame rate. By changing the implementation to C++, we hoped that it can display a lot faster.

Tool	Frame rate
Java CPlaneDisplay with ERS210	5-6 frames/sec
Java CPlaneDisplay with ERS7	3-4 frames/sec
C++ CPlaneClient	24-30 frames/ sec

Table 5 - Performance improvement of C++ CPlaneClient

The experiment showed that C++ could be 10 times faster than Java in displaying CPlane. This work was later on developed into full feature client whose screenshot is shown in Figure 44.

The GUI is handled by Qt library, a full fledge GUI library used by open source community. The library itself provides a complete development environment, from a GUI designer to make tools. For example, the designer creates forms, and put the information into a project file, say CPlaneClient.pro. Another tool called 'qmake' convert the project file into Makefile. All the developer needs to make the program is to type "make" afterward. For more information of Qt library, see [20].

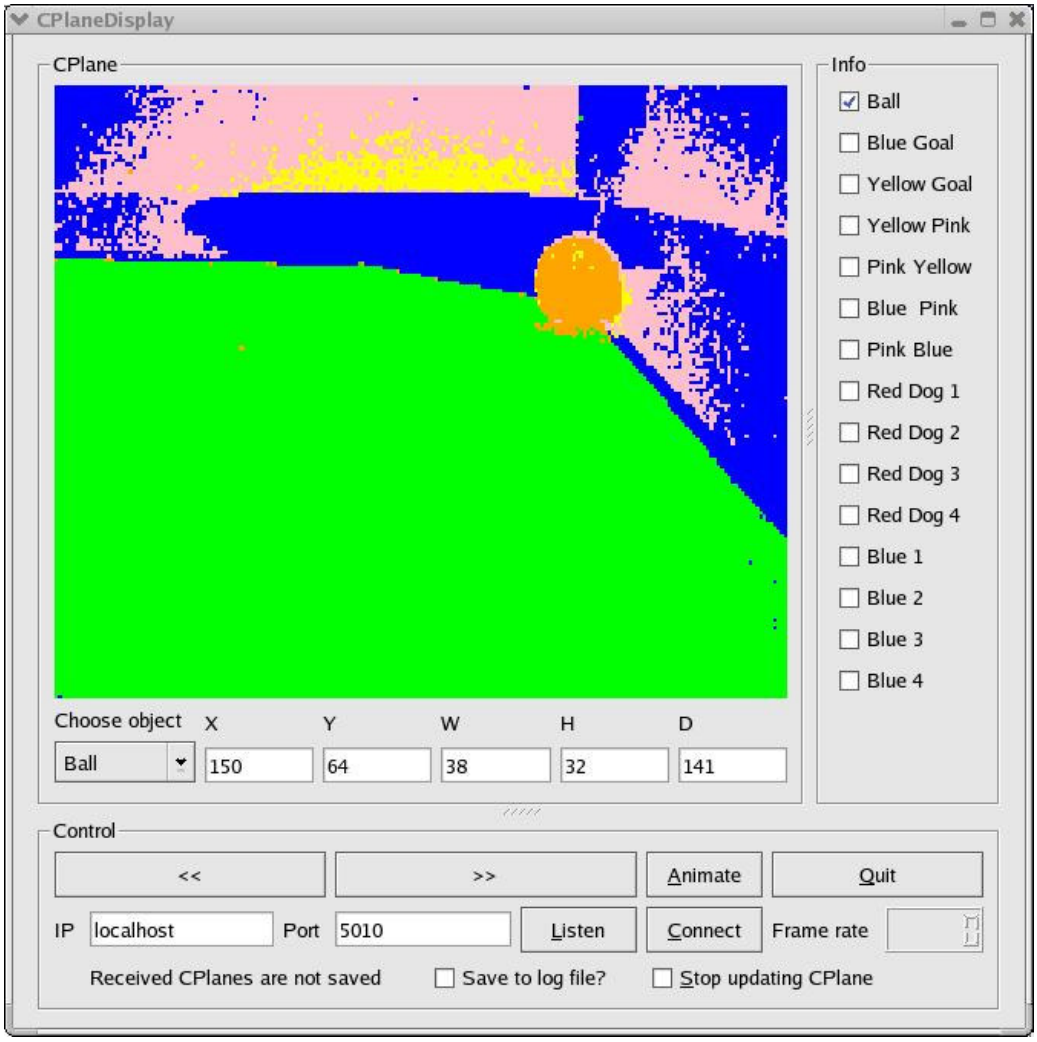

Figure 44 – A Screenshot of C++ CPlaneClient

The recognized objects are shown on the right panel, along with their details information (coordinates and size) in the bottom of the CPlane. The client can also log the cplanes. However, after implementing this, we found out that logging was also done in base station client, which is the first application that receives data from the dog. Therefore this feature is unfortunately never used.

Animation is also implemented in CPlaneClient. It can read cplane log files, and animate them, similarly to Java CPlaneDisplay.

6.2. JointDebugger

At the early stage of porting code to the new ERS7 dog. We got a problem of "Battery-Over-current". This problem happens very frequently when the dog performed improper walking steps (bad walking style causes the dog to perform action that is over its ability, and thus increases battery current to the limit). When "Battery-Over-Current" occurred, the dog shut down itself automatically. This tool is created to find solutions for that problem, aiming at detecting such improper steps. To detect such conditions, information of current joints angles,

and the command that we tells the dog to do needs to be compared. The JointDebugger is created to send all those information to the base station for analyzing. A screenshot of the tool is given below:

Figure 45 - Joint Debugger screenshot

The tool is finished. However, after analyzing such information, we could not find any particular pattern that caused the shutdown. The cause of the problem was lying in the early version of OPENR system, which was fixed in the later versions.

The tool was not successful in term of the original purpose; however, it is still a necessary tool that could be used in the future. For example, a form of stuck detection using the same information that this tool provided has been developed [8].

7. Future works and conclusion

Robocup project has been developed into a relative large scale project, which involved many smaller projects. This report only showed small parts of it. Even though, there are a lot improvements can be made in the future works.

Future improvements have been partially addressed in each chapter above. However, there are still areas that never been explored. We will present some of them here.

We could create more motions for the dog, especially the kicks. The shortage of kick has restrained our performance in the open challenger as well as the main game. In order to do this, a motion editor could be made, allowing visually and practically edit a dog motion. By visually, we mean the motion can be previewed in a 3D model without the need of playing it on the dog. By practically, we mean once the draft motion is created, it can be fine-tuned, adjusting on the dog. Instead of viewing on 3D model, each step of the motion can be transferred directly to the dog, and showed immediately. Such editor has been used by some teams, and can practically copy any motion from human's eyes.

Vision is a big area of improvement in our system. The lack of "quality insurance" tools may prevent any team from performing at their best in the competition, even though, it works well in labs. Our methods of detecting visual objects by color blobs can also be combined with other methods of detecting visual obstacle [21], which is less sensitive to light condition.

On the other hand, other methods low level vision can also be experimented. New way of learning color calibration may be developed, which is more robust and easier to use. One of them is incremental learning method, has been developed many years in our school. Incremental learning is well-known for its ease of maintain, and wide availability of approaches as well as tools.

Data fusion of opponent localization could also be improved. This year, opponent filter is not used, because of several reasons 1) it is not used in strategies and 2) dog visual detection is not good enough. However, having good localization of opponents means a total control of the game situations and the strategies would be a lot more intelligent.

The improvement list is almost endless. That proves the great vision of the project "to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined..." robocup.org. We believe that the field of RoboCup researches will be growing up more and more. The project will definitely be seeing great researches, ideas and achievements to come.

References

1. J. Chen, E. Chung, R. Edwards and N. Wong. rUNSWift 2003 Report.

2. G. S. Hornby, M. Fujita, S. Takamura, T. Yamamoto, and O. Hanagata. Evolving Robust Gaits with AIBO. In IEEE International Conference on Robotics and Automation, pages 3040–3045, 2000.
3. Bernhard Hengst, Darren Ibbotson, Son Bao Pham, and Claude Sammut (2001). Omnidirectional Locomotion for Quadruped Robots. In A. Birk, S. Coradeschi,8 and S. Tadokoro, editors, Lecture Notes in Computer Science, RoboCup 2001: Robot Soccer World Cup V,pages 368–373. Springer, 2002.
4. David Wang, James Wong, Timothy Tam, Benjamin Leung,Min Sub Kim, James Brooks, Albert Chang, and Nik Von Huben. The UNSW RoboCup 2002 Legged League Team. Undergraduate thesis in computer and software engineering, University of New South Wales, 2002.
5. M. S. Kim and W. Uther, "Automatic gait optimization for quadruped robots," In proceeding of ACRA2003.
6. N.Kohl and P. Stone, "Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion", In Proceeding of ICRA2003.
7. Seperation of Concern: http://c2.com/cgi/wiki?SeparationOfConcerns , Wikipedia.
8. J. Hoffmann and D. G¨ohring, "Sensor-Actuator-Comparison as a Basis for Collision Detection for a Quadruped Robot," in 8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences), Lecture Notes in ArtificialIntelligence, Springer, 2005. to appear.
9. Melanie Mitchell, "An Introduction to Genetic Algorithms"
10. J.C.Z.Montealegre, J. Ruiz−del−Solar. UCHILSIM: A dinamically and Visually Realistic Simulator for the RoboCup four Legged League, in 8th International Workshop on RoboCup 2004.
11. Learning to Play Keepaway http://www.cs.utexas.edu/users/AustinVilla/sim/keepaway
12. Genetic programming to learn Keepaway Soccer http://www.cs.nott.ac.uk/~smg/keepaway.html
13. Videos of OpenChallenger skills
14. University of Pennsylvania Robocup team report, 2003. http://www.cis.upenn.edu/robocup/source/UPenn03.pdf
15. Nicodemus Sutanto, Undergraduate Thesis Report.
16. Eric3 IDE : http://www.die-offenbachs.de/detlev/eric3.html
17. Python common object structure: http://docs.python.org/api/common-structs.html#l2h-824
18. Python extension writing http://starship.python.net/crew/arcege/extwriting/pyext.html#module
19. rUNSWift code online documentation. http://cgi.cse.unsw.edu.au/~tomv/cgi-bin/robocup/wiki.pl
20. Qt documentation/tutorials: http://doc.trolltech.com/3.3/index.html
21. J. Hoffmann, M. J¨ungel, and M. L¨otzsch, "A Vision Based System for Goal-Directed Obstacle Avoidance used in the RC03 Obstacle Avoidance Challenge," in 8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences).
22. K.C.Pham, Summer Research 2004 technical report. https://.../trunk/papers/TechnicalReports/KimSummer2004Report.pdf

23. C.K.Lam (Daniel), Undergraduate Thesis Report.
https://.../trunk/papers/Thesis2004/DanielThesisB.pdf
24. Martin L¨otzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias J ungel, Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL, ”in 7th International Workshop on RoboCup 2004 (Robot World Cup Soccer Gamesand Conferences).

Appendix

1. Dynamic Architecture example:

BallHoldingLocalisation is an example of using so-called dynamic architecture, by connecting sub actions together and reconnecting them on-the-fly. Connecting two sub-actions means setting one to be the outState of the other, so that when one sub action is done, it will execute the next action.

2. Might includes helping players/tools:
   - LatencyTester: This player is to measure the latency of the network. It requires two dogs numbered 1 and 2. Dog 1 will send a numbered token n to dog 2. When dog 2 received token n, it responds back number n. When dog 1 receives token n, it continues sending token (n+1). The latency is then calculated from the time dog 1 sends token n, until the time it receives back token n.
   - ReadJointPlayer: This is similar to the old remote player. It responds to various base station commands, to execute various actions.
   - Odometer Calibration: This player is used to work out odometry information of a walk. The walk command is sent to the dog, the human waits until the dog makes a full circle, and comes back the original position. The diameter of the circle is calculated, along with the number of frames it took to complete the circle. From that information, it is able to work out how much forward/left/turn it traveled for one frame.
   - Odometry Learner: This player is to attempt to measure odometry automatically. It does so using GPS. It starts off by going to a position on the field. The walk command is executed for a number of frames. Using GPS, the trajectory that it has gone is logged, from which the odometry information is calculated. However, it is realized that the current GPS is not accurate enough for this to work.
   - Sanity check player: LandmarkTester. This player is used to send CPlane display “on demand”. For example, sending CPlane about ball is controlled by a variable: expectBall. If expectBall is True, then the dog would send every cplane that it does not see the ball, in which case the ball is missed. If expectBall is False, it would send cplane that it contains ball in which case a phantom ball is recognized. The variable can be set to None to not send any cplane at all. Similarly, variables to control beacons, goals, robots are used to send bad vision cplane.