Soccer Behaviors on Bi-pedal NAO Robots.

Arumae Kristjan, Myhre Sarah, and Olschewski Cassian.

School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida.

The project encompasses the framework and setup needed to run soccer behaviors on the NAO robot. The behaviors are constructed using the NAOqi API provided by Aldebaran Robotics. The framework is developed dynamically to accept any sort of working model for robot behavior that runs cyclically. Along with the code and framework the project also gives developers a through walkthrough for setting up the software.

***Index Terms* — Robotics, NAO, Webots, Artificial Intelligence.**

I. Introduction

Using soccer behaviors as a medium for robotics competition has now existed for more than a decade. It is now an international competition with universities across the globe participating to develop the most robust working team. Currently, however there exists no team for the University of Central Florida. This project was a chance to kickstart interest in competition as well as Artificial Intelligence in general. The project included simulation of code as it would appear on the robots with no physical robots present in the scope of the work.

The development touches on behavior via behavior-trees as well as finite state machines. Use of computer vision is discussed alongside locomotion control and modification.

II. Related Works

Preliminary research came from several sources. First and foremost was the technical review of the CMUnited team performance in the 1998 Robocup competition[1]. The article discussed behavior modeling and decision weighing as it pertained to Carnegie Mellon winning the RoboCup that year. Also included is the Advanced Robotics with Artificial Intelligence for Ball Operation(ARAIBO) technical works[2]. The team is from Tokyo and discussed their dynamic programming approach to building a successful team. Also included was a conference paper on Policy Gradient Reinforcement Learning for Optimizing Arm to Leg Coordination and Walking Speed. This discussed machine learning and synchronization of multiple joints for movement optimization.

III. Software and API

The project utilized several key pieces of software. For development Eclipse, and Sublime Text were used with NAOqi support for Python. Simulation ran from Webots 7.1.1.

The NAOqi APi was provided via Aldebaran Robotics for use with their hardware-the NAO robot. The API is available in several languages with the most prominent being C++ and Python. The API works with brokers loading proxies for method calls onto the robot. Each method call, either blocking or non-blocking, controls one of several proxies. The most common proxies are for memory, locomotion, and posture. Each proxy call is made by assigning the object the IP and Port of the robot the code is loaded on.

Working with the API was done initially on Eclipse. Eclipse has a well-stocked list of plug-ins one of which is Pydev. Pydev is for easy Python development within this IDE. Other than the syntax and highlighting, Pydev also provides runtime support such as breakpoints. Sublime worked as a fantastic editor for Python as well in Ubuntu, and running code from it was simple.

The Python interpreted scripting language was our language of choice due to the natural structure of the language. Python has various mechanisms that facilitate quick implementation of functionality. Dynamic typing in Python allows developers to create and then maintain short, readable code. This makes the developer's job easier by placing the tasks of declaring, tracking, and, checking data types into the hands of the interpreter. Just this mechanism alone makes Python code shorter and more readable than other static typed languages such as Java and C++. In addition, the built in Python libraries are written with short symbol names to couple with the idea of shorter code bases without creating unreasonably obfuscated naming conventions.

Another mechanism, forced indentation, follows the idea of short but readable code. The indentation of nested expressions is important in producing readable Python scripts. The indentation requirement that Python enforces effectively reduces the room for error when a developer is working with any given code base. Further, the

requirement makes code produced generally more readable and thus more maintainable.

IV. Framework

Taking advantage of Python's clean-code mechanisms, the framework itself was designed with maintainability in mind. Due to the nature of this project, the requirements for soccer player behaviors are vast, and as with any artificial intelligence, functionality is key. Knowing this, our design choices consisted of those that would facilitate the addition of new functionality in future times.

We began the framework design process by analyzing the problem domain. Our scripts needed to be able to instruct a humanoid robot to play a game of soccer against other robots of the same type. Thus, the creation of the framework was driven by the existence of two main sub problems in the overall problem domain: One, how can the project be divided amongst the members of the team? And two, how can we logically implement the desired design functionality?

To answer the first problem we must look at what objects are present in the system and what responsibilities those objects have. A good place to begin is with the main agents in our soccer game, the players. A robot competitor is an obvious choice for encapsulation. The robot itself is a rather complex and general object. With the current sub problem in mind, is there a way to logically break down the robot into programmable modules? Now we can see that the sub problem reduces to one of design modularity when viewed in a problem-specific light.

The next step was to look at what the robots were logically composed of so that they could be more easily constructed in code. The robots are humanoid, and do have functioning vision and limb components. Coincidentally, those two components were crucial for carrying out a successful soccer game, virtual or not. We encapsulated the functionality of the legs and eyes of the robot and created modules to represent those function sets. Lastly, we ask, what drives these robot components? In the robot players, the scripted soccer behaviors had to be able to delegate commands to the robot's hardware components, or in our case the simulated components. Upon encapsulating the logic models we noticed that the soccer behaviors themselves create has-a relationships with the robots. This discovery was critical in the furtherment of our design. The has-a relationship that each subcomponent makes with the robot guided our design when modeling the relationships between the objects identified for encapsulation and implementation during the initial design process.

Ultimately, the need for a modular design due to the nature of team based projects caused us to devise a domain model that reflected, and facilitated the development of programs that were to be created in a modular manner. Now that we had a solution to the first sub problem, and incidentally a feasible concept for an extensible, modular, and maintainable framework, we could begin to construct a solution to the second sub problem, how can we logically implement the desired design functionality?

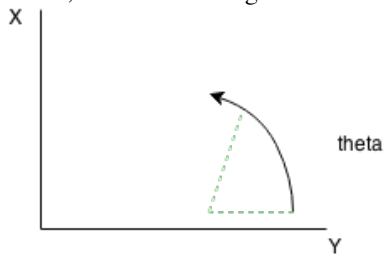
Taking the natural relationship of the objects identified earlier into consideration, the most logical design pattern to use was the composite pattern. As the name suggests, it is largely based upon the concept of composition. The Composition Design Principle places the general concept of composition into the light of object-oriented design in software. Conceptualized by the authors of "Design Patterns: Elements of Reusable Object-Oriented Software"[3] the composition design principle attempts to address the question of when to model is-a relationships versus has-a relationships in code.

In our design, the robot is composed of RobotEyes, and RobotLegs classes. These objects contain methods for interfacing with the open computer vision libraries, and NAOqi libraries respectively. The complexities of the robot players are decoupled from the robot class itself which allows for multiple team members to implement the robot scripts at once. The robot behaviors were encapsulated into their own classes and each one extended the LogicFor class to implement specific soccer behaviors. The LogicFor functionality was uniformly controlled by the FSMWrapper class, which handled the lifetime state of the Threads used to execute the instructions of the behaviors. All of the components were unified under one centralized class called the TeamBuilder. This class provides the entry point to the program for the host computer and delegates the commands to start behavior threads on the physical or simulated robots via network communication methods implemented by the NAOqi framework.

V. Behavior Evolution

The behaviors in the project built upon one another. Meaning that simpler ones were created and from them evolved more complex orderings. More importantly the simpler behaviors became building blocks to be implemented in fuller working modules. The following section describes the simple chasing module and then the implementation of a full working offensive behavior. Both of these utilize the framework described in section 4.

From the very beginning it was clear that giving a robot the same method call with no adjustment would result in unsatisfactory results. The API includes more than 10 methods just for walking. These methods take in parameters as simple as the desired X and Y coordinate, as well as more complex lists of parameters to define all joint control. After testing these methods we noticed that making several calls to moveTo()-a method we used to walk forward-resulted in the robot veering to a side from time to time. The robot would also often lose balance and tip over. The method did include a theta value which should control the curvature of the walk in the direction desired, as seen in the figure below.



The robot views X as the forward axis and any movement sideways as on the Y axis. This angular change should be accounted for during every cycle of movement since slight differences will occur.

Another element added to every walk was arm-swing. The robots were naturally very stiff unless otherwise instructed. This caused the torso to not move when the lower body did thus causing the robot to quite often tip to the side. Although the behavior trees do account for the occurrence of falls it does slow down performance significantly to have to get up often.

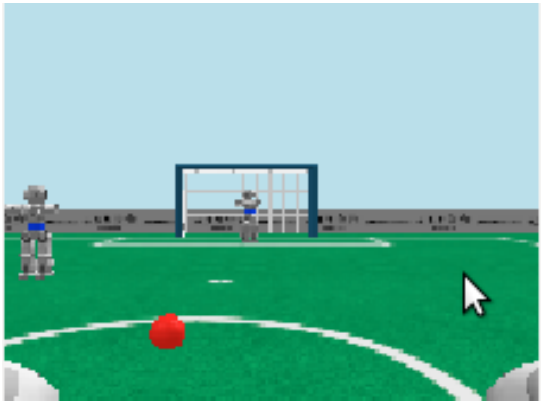
The code below shows the fully parameterized version of the method call to walk along with a chart to illustrate what each value corresponds to.

```
36      def walk(self, value):
37          self.motionProxy.setWalkTargetVelocity(1.0,0.0,value,1.0,
38                  [#LEFT
39                  ["MaxStepX", 0.07],
40                  #maxStepYRight,
41                  ["MaxStepTheta", 0.349],
42                  ["MaxStepFrequency", 1],
43                  ["StepHeight", 0.015],
44                  ["TorsoWx", 0.0],
45                  ["TorsoWy", 0.0]],
46                  [#RIGHT
47                  ["MaxStepX", 0.7],
48                  #maxStepYLeft,
49                  ["MaxStepTheta", 0.349],
50                  ["MaxStepFrequency", 1],
51                  ["StepHeight", 0.015],
52                  ["TorsoWx", 0.0],
53                  ["TorsoWy", 0.0]])
54
```

Name		Default	Minimum	Maximum	Settabl
MaxStepX	maximum forward translation along X (meters)	0.040	0.001	0.080 [3]	yes
MinStepX	maximum backward translation along X (meters)	-0.040			no
MaxStepY	absolute maximum lateral translation along Y (meters)	0.140	0.101	0.160	yes
MaxStepTheta	absolute maximum rotation around Z (radians)	0.349	0.001	0.524	yes
MaxStepFrequency	maximum step frequency (normalized, unit-less)	1.	0.	1.	yes
MinStepPeriod	minimum step duration (seconds)	0.42			no
MaxStepPeriod	maximum step duration (seconds)	0.6			no
StepHeight	peak foot elevation along Z (meters)	0.020	0.005	0.040	yes
TorsoWx	peak torso rotation around X (radians)	0.000	-0.122	0.122	yes
TorsoWy	peak torso rotation around Y (radians)	0.000	-0.122	0.122	yes
FootSeparation	alter distance between both feet along Y (meters)	0.1			no
MinFootSeparation	minimum distance between both feet along Y (meters)	0.088			no

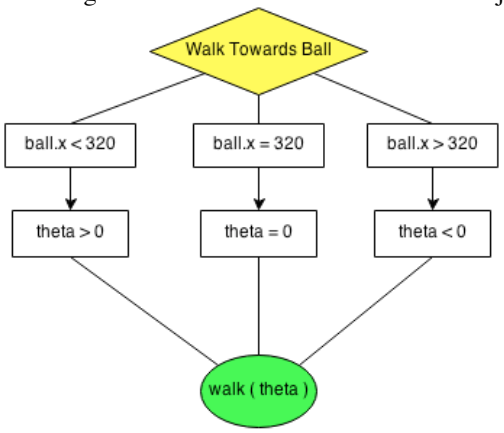
If any parameter is left blank it takes the default value. The values set for this walk almost eliminated any tipping over and resulted in a smoother walk while tracking. The walk was also slowed down to avoid any difficulties.

Having a method call that provides better results was not enough however. To make the walk more optimal changes must be made every cycle to slightly adjust trajectory. To do this the theta must have a relationship to whatever object is being tacked in the field of vision. To illustrate this we will use the ball. The view below shows the robots head camera with the ball in view



The resolution of the image is 640 by 480 thus the middle line of the image is 320. If the returned coordinates of the

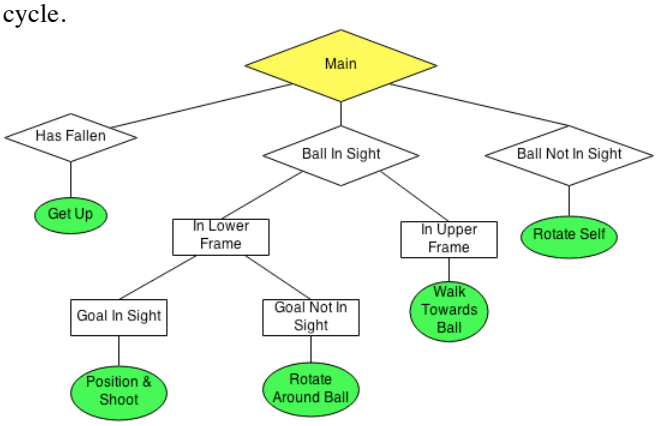
ball are below 320 then the value that will be theta is calculated from the bottom of the frame clockwise to the center, otherwise theta will correspond to the angle created from the bottom of the frame counter-clockwise to the center line. Wherever the ball is results in where the angle stops. This value is then negated if the robot must move clockwise and positive otherwise. The angle is converted to radians. This value is then scaled down to twenty percent and passed in as the theta value. The reason for the scaling is to not cause the robot to move too severely resulting in a smoother recalculation of the trajectory.



This algorithm produced the most desired results. The cycling of the walk trajectory must also take into account the wobble of the frame. At any point the returned image might be tilted one way or another. This is also a reason for scaling down the theta to a fraction of its true value. In earlier implementations when the parameter was the true theta- with a ceiling of 1rad and a floor of -1 rad-the robot would start doing very sharp turns to correct the path, this would cause the ball to be lost as quickly as it was located.

The goals for having an all-around working offender was to have a behavior that would not interfere with other players' behavior and also not need assistance to carry out its own tasks. The logic behind the offensive player is based on the principle of having a way to get out of any un-ideal situation, in other words not getting stuck. The complexity of the offender should also be mature but simple. There should be no point in the code where the robot cannot get back out to an earlier level due to contradicting decision parameters. Therefore every level must operate smoothly to avoid any sort of cyclical dependencies. That being said the maturity should not only resolve trivial cases thrown at the robot but rather be dynamic and operate in a multitude of scenarios.

The figure below illustrates the logic given to the offender. Above all else the player is concerned with having the ball in view, and then having access to it. It is important to note this applies to the robot when it is in a standing position. The tree does account for falls every cycle.



Starting from the top of the tree the robot must circle until it finds the ball. This rotation should only take seconds for a full scan of the field. If at any cycle the ball becomes present in the head or the torso camera the rotation is killed and the ball is either approached or the goal is located. For the former scenario the algorithm described in the walk section is used, with a new theta calculated each time. The robot will in this case be facing the ball head-on as it disappears from the upper frame into the lower one. During testing it was clear that if the ball was close enough to be in the lower frame the robot should start positioning to score or pass. This gives a metric as to how far away the robot is from the ball and the threshold is always constant.

With the ball in the lower frame the robot would start to position themselves to align with the goal. This motion keeps the ball in front of the robot until the goal comes into view. The goal must be within a range of the center of the screen. This range is quite wide since the goal itself accepts a spectrum of kicks. Another reason for not having a small window is so that the robot will carry out an action without having to re-position due to a slight miscalculation. A percentage of error is to be expected for the kick to the goal. When testing with smaller range a robot make take hundreds of cycles before it is confident enough to kick and score a goal. Although this previous method provides a more accurate end result, the chance of interception or other ball loss becomes increasingly higher.

After alignment the player will do a final approach to the ball. This approach utilizes the same walk algorithm as before but uses a slower walk speed to keep the robot steadier. Specifically the speed is 26% of the normal speed. During every cycle at this point the robot makes sure that the goal is still also in an acceptable range and

then moves forward. This being so far down the behavior tree makes the behavior the most precise.

At the very end of this behavior tree the robot should be at the ball, ready to kick. If the ball coordinate returned is more than 460(of a total of 480) pixels in the Y directions from the top, then the robot is close enough to attempt a kick. This takes into account another aspect of the cycles. More than likely the robot will finish the last step from the previous cycle in which they were already moving, bringing the ball even closer to the foot of the player. The code calls a type of stop-movement that lets the previous call return to a steady state. This allows for the player to be stable by the time the next call is made. Otherwise the player may be on one foot, tilted, or even having fallen. After the kick the robot will once again chase the ball and complete the action again. There is also a chance that the ball gets just close enough, but also out of range, in this case the robot will of course start to look for it again.

This behavior does not account for any saving of world states. For example if the robot does loses sight of the ball in the right edge of the frame he will still rotate left-as is default. By implementing a behavior tree rather than a finite state machine (FSM) the robot would not have a decent way of keeping track of other players and objects outside of the frame of vision.

VII. Conclusion

The project delivered several working behaviors that encompassed a complete team. This code and the framework that goes alongside of it is available to Dr. Sukthankar as teaching material for future robotics classes, as well as any graduate level team development. With that is information regarding

Acknowledgement

The authors wish to acknowledge Dr. Gita Sukthankar, Dr. Mark Heinrich and Astrid Jackson for their guidance and support of the project.

References

[1] Stone, Peter, Manuela Veloso, and Patrick Riley. "The CMUnited-98 champion simulator team." *RoboCup-98: Robot soccer world cup II*. Springer Berlin Heidelberg, 1999. 61-76.

[2] Takeshita, Kazutaka, et al. *Technical report of team araibo*. Technical report, ARAIBO, 2007.

[3] Gamma, Erich, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.