Northern Bites
Team RoboCup 2009

Northern Bites 2009 Team Report

Tucker Hermans, Johannes Strom, George Slavov
Jack Morrison, Andrew Lawrence, Elise Krob
Prof. Eric Chown
Department of Compute Science
Bowdoin College
8650 College Station
Brunswick, Maine, 04011 USA
echown@bowdoin.edu
http://robocup.bowdoin.edu

February 26, 2010

Contents

1 Introduction

The Northern Bites is Bowdoin College's entry to the Standard Platform League (SPL) of RoboCup. Our team is one of only a few teams comprised solely of undergraduate students, and the only competitive entrant from a Liberal Arts college. Despite coming from a small school of about 1700 undergraduates, we have performed well in every RoboCup since our debut in 2006. During that first year, we competed using the Aibo robot dogs, at the US OPEN, and then at the RoboCup championships in Bremen, placing 10th. The next year, we placed in the top three at the German OPEN in Hannover, and finished first at the 2007 RoboCup in Atlanta, Georgia. In 2008, we were the only undergraduate team to compete both in the Aibo league and the inaugural Nao league. We placed 3rd in the Aibo league after losing in penalty kicks to the German Team in the semi finals, and made it to the quarter finals with the Naos. In 2009, we struggled at the US OPEN, which we hosted, but made big gains before the RoboCup World Championships in Graz, Austria, where we placed second.

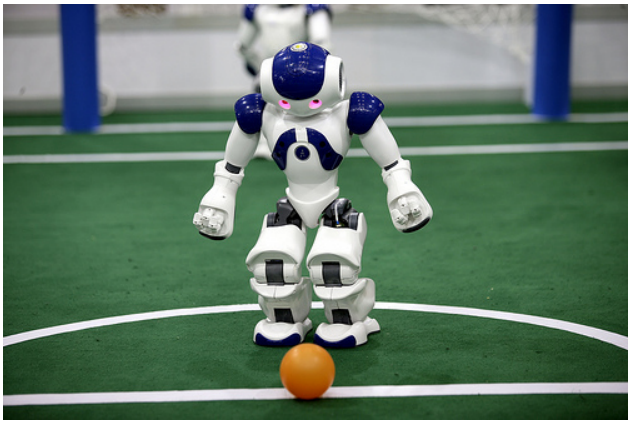

Figure 1: A Northern Bites Robot at the 2008 RoboCup in Suzhou.

Our team has continued to achieve high quality soccer play despite our small team size and high turnover rate. Our program's focus on education and our "fire" to succeed have enabled us to perform at a high level, especially on the Nao robot, where many teams struggled to even walk, let alone score goals.

One major change in the team's philosophy this year was to move our code base to the Git Version Control System. Aside from providing a superior development experience to the Subversion system we used in previous years, we also made our Git repository publicly available. This is in accord with the spirit of RoboCup which is about scientific cooperation and constantly making progress as a league. However, we are a proud team and recognize that RoboCup is still a competition so we moved our main development repository offline for our last two months of development leading up to the competition. It is now back online and can be found at `http://www.github.com/northern-bites/nao-man`.

The remainder of this document describes our team strategy and the software systems which we entered at RoboCup 2009 in Graz, Austria. We start with a description of our perceptual system, leading into a discussion of our localization methods, and an overview of our walk engine. Finally, we discuss how these subsystems are coordinated strategically.

2 Vision

Our vision system changed very little from 2008 [2]. Mainly we adapted our old system, which was built for the Aibos over to the Naos. This resulted in a much leaner system since we were able to drop Aibo-specific code involving things such as chromatic distortion which are not a problem on the Nao. The Nao also afforded us with a chance to simplify in other ways.

Whereas the Aibo's head could tilt rather dramatically a great deal of attention was paid to the angle of the head and how that impacted our vision algorithms. This is no longer the case for the Nao, and any angle offset comes mostly from the hips. To simplify our vision in our initial scan of the image we do a straight vertical scan from the bottom of the image up. We do not, however, scan the entire image.

First we do some initial scanning to get a rough outline of where the field is. We then use this information in conjunction with a pose estimated horizon to bound our scanning algorithm. If we appear to be scanning a relevant object such as a goal, we ignore the bound. The field boundaries also serve as useful information for doing sanity checks on any potential objects. For example, balls should never be detected anywhere but on the field. Goals, on the other hand, should occur at the edges of the field.

In terms of goals we simply look for the longest runs of Blue and Yellow color that we can find. Once we have completed scanning we do an active search for the goals at the point of those long runs. This process is basically identical to what we did in 2008 [2]. The one embellishment we added this year was the ability to correct for the fact that the goals will not always appear perpendicular with respect to the ground (Fig. 2). We detect this by scanning out from the corners of a candidate goal. If the goal is skewed enough we essentially through out the pose generated slope of the image and calculate a goal specific slope based upon the actual shape of the goal.

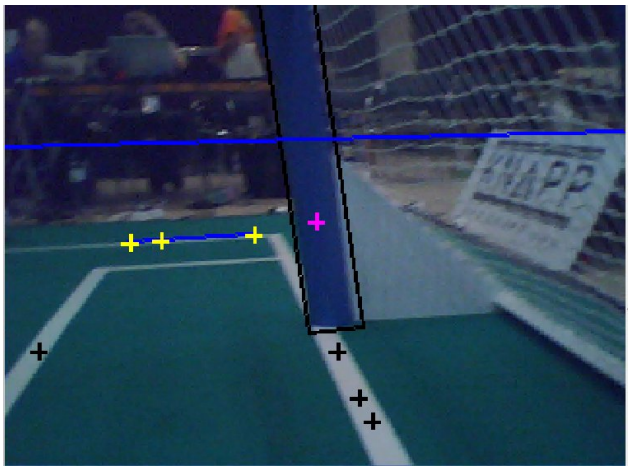

Figure 2: **Skew Detection** The angle of the post is sometimes sharper than the pose estimated slope shown in blue. The black rectangle around the post is the result of adjusting the expected angle on the goal.

We also added a new field object this year – the crosses at either end of the field. Our detection method is very crude, but surprisingly effective. As we scan the image we run length encode blobs of white color. To reduce the amount of coding we limit the runs to be only in sections of the

image on the field and only within parametrized size bounds. After scanning we look through all of the candidate blobs and simply start throwing out all of the blobs that couldn't be crosses. For example, parts of lines are thrown out, blobs that aren't generally square, blobs that aren't completely surrounded by green, etc. At no point do we actually try and determine that the shape of the blob is a cross.

We experimented with robot detection this year but found that even with the improved cameras on the Naos it is still not easily accomplished. The main problems have to do with color separation. For example, in Austria the blue uniforms of the Naos were almost identical to the color of the goals. Nevertheless this is one of our primary objectives moving forward.

Our work on field lines this year involved mainly adapting to using the Nao's and improving how we identify field intersections. In 2009 we only used intersections to help localize and in 2010 our team's number one goal will be to fully incorporate all field line information into our localization system.

3 Localization

A crucial element of soccer strategy is having an accurate estimate of the field positions of all the players. Using the various field features detected by our aforementioned vision system, we use a standard 3 degree of freedom Extended Kalman Filter. Particularly, we use range, bearing estimates to goal posts and corners as landmarks.

One of the most difficult aspects of localization in the SPL is solving the data association problem. That is, given an ambiguous observation to an L corner, data association is the problem of establishing a match to the correct L corner on the field. We found it sufficient to simply do a euclidean nearest neighbor matching to make data associations.

The Jacobians needed for integration of just range bearing sensors into the EKF are relatively straightforward, and are discussed, for example, in [8].

To develop our EKF localization module, we implemented an offline simulation environment which allowed fake landmark observations to be detected. While this was not a high fidelity simulation, we were able to use it to accelerate our progress.

In addition to modeling the robot's position, we also filter the position of the ball, including its expected dynamics. Since the location of the ball is so central in soccer, it simplifies the state estimation if the ball is treated separately from the pose, and any cross-correlation between the robot pose and the ball is ignored.

Finally, an in-depth treatment of the localization system can be found in [4]. Looking forward, expanding our EKF framework to correctly integrate measurements to lines and the center circle will allow us to increase our positional accuracy and would expand our soccer abilities significantly (possibly passing, more accurate goalie saving).

4 Locomotion

For humanoid robots, a crucial part of playing soccer is locomotion. Because of the inherently unstable dynamics of the robot, and the concern for the safety of the machine, we are especially driven to create a walk engine which is robust to external disturbances yet still can move quickly enough to participate in an engaging soccer game. Unlike most of the other teams in the SPL, we wrote our own walking engine from scratch. Based on previous research in humanoid locomotion,

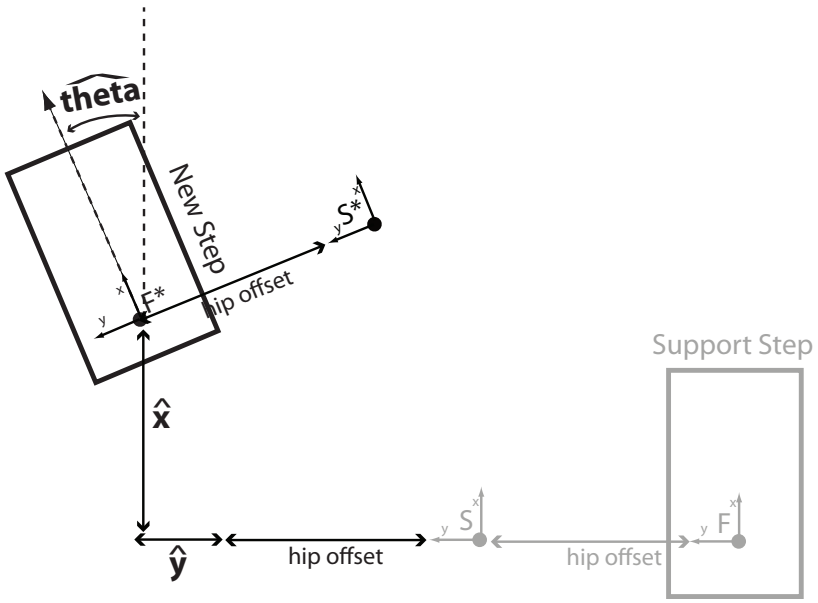

Figure 3: An additional step is being added using a motion vector $(\hat{x}, \hat{y}, \hat{\theta})$.

including by some SPL teams, our approach uses a ZMP model of the robot coupled with a preview controller to generate walking trajectories. Our approach is detailed in [6, 7], but we give an detailed overview of the salient pieces here.

The remainder of this section discusses how are system breaks successively breaks down the walking problem into sub problems which can be solved more directly.

4.1 Step Planning

A natural representation for humanoid walking is a sequence of discrete steps. These steps explicitly constrain the locations of the robot's feet at certain times. The steps also provide an indirect constraint for the path followed by the robot's center of mass, if the robot intends to maintain its balance.

Our engine provides a variety of methods for generating these steps, enabling omnidirectional motion. Most generally, we consider a request to bring the robot to a specified velocity. Since soccer is confined to a plane, it suffices to represent this as translational velocity in the forward and lateral directions, and as a rotational velocity about the vertical axis: $\mathbf{v} = (\dot{x}, \dot{y}, \dot{\theta})$. Using this interface, the robot can be controlled by sending velocity requests asynchronously. For example, to chase a ball, the robot can be instructed to walk towards the ball by setting the velocity in the direction of the ball.

For finer grained control, we also allow requesting a specific number of steps to be taken at a given velocity. This is especially helpful when approaching the ball at close range.

A list of proposed steps are maintained in a queue, and until a step has been "committed to" (see below), a new motion request can supersede the current plan. For motion far in the future, steps are not created, instead the desired velocity is stored, and new steps can be generated as needed. Given a velocity vector as described above, step generation can be visualized as in Figure 3.

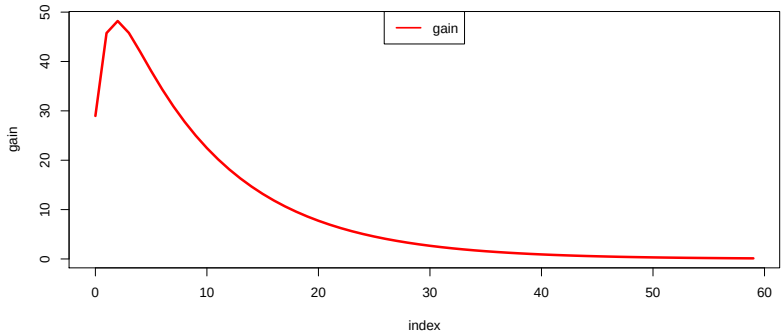

Figure 4: The weight associated with future motion plans. Intuitively, upcoming motion is weighted more heavily than motion in the far future.

4.2 ZMP Preview Control

The next layer of complexity below step placement is movement of the center of mass of the robot relative to the legs (or vice versa). To achieve an elegant solution to this, we build on control theory most recently advanced by [3] which is called preview control. The preview controller directly encompasses the idea that to some extent, humanoid walking must "preview" the future motion plan in order to generate a dynamically balanced trajectory. In this case, the locations of some of the future steps must be fixed some time before the steps are actually taken, forcing us to "commit" to stepping in certain places. This period of look-ahead is called the preview period, and for us was about 1.4 seconds. This preview window can be visualized in Figure 4 nicely by looking at the internal weighting the controller associates with future motion. The exact integration of this weighting is discussed in more detail in [7].

The preview controller also depends crucially on the ZMP concept which has been widely applied to the humanoid walking problem. The ZMP provides an analytical metric to determine the balance of the robot. In essence, it measures the location on the surface of the foot where the ground normal force must act to keep the foot from rotating with respect to the X and Y axes. By planning to keep the ZMP in the center of the foot (e.g. as far away from the edges of the foot as possible), the controller can plan motions of the center of mass which will keep the robot from falling. In turn, the planned sequence of steps provides a method for generating the ideal, or reference path for the ZMP. As the support foot switches during the gait cycle, the planned reference ZMP is passed from the current support to the future support foot. This process is visualized in Figure 5.

Although the ZMP can be determined analytically, considering the full dynamics of the robot is too computationally complex to compute online. Instead, the robot is modeled as a simple cart of a massless table, where the mass of the cart and the robot are equivalent (See Figure 6. This allows computation of the optimal preview controller to be straightforward and tractable.

4.3 Inverse Kinematics

Once the preview controller can specify the location of the center of mass relative to the support foot, the angles for the support foot can be solved using Inverse Kinematics. Initially, we used an iterative damped least squares non-linear optimization method to compute the required angles

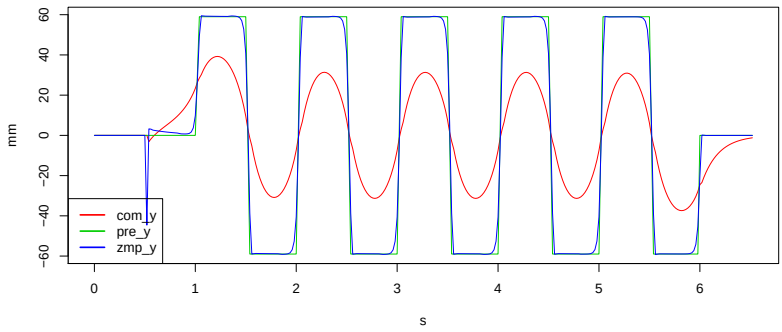

Figure 5: The ZMP reference, and resulting Center of Mass path over time, in the lateral direction, for a sequence of ten steps.

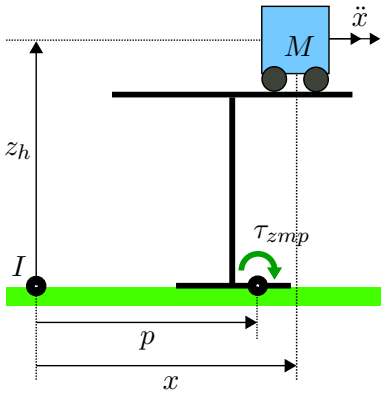

Figure 6: The simplified cart table model for a robot which allows the ZMP to be computed easily.

given a 3D coordinate for the end effector. Much more efficient is to compute the solution in closed form, the method for which was graciously provided to us by Prof. Daniel Lee of the Pennalizers.

For the swinging leg, the 3D end effector destinations are computed by interpolating the source and destination steps.

4.4 Gait Tuning and Optimization

What we have just discussed is a broad overview of our motion engine. Unfortunately, the realities of the NAO robot platform and the inherent difficulty of humanoid walking require considerable tuning beyond the basic conceptual implementation. In practice, our motion engine was parametrized by over 30 knobs. Many of these were necessary parameters, even in theory. For example, the height of the robot's body during walking, as well as the duration of each step. In reality, only six or seven parameters need to be adjusted when moving to a new carpet or floor surface. Some of the other parameters, although static, were crucial to the functioning of our walk. A full list of these parameters is available in the source release, in the README in the motion folder. However, some of the more important parameters are discussed below.

Perhaps the most crucial observation is that lowering the stiffness of the joints increases the robustness of the walk (and lowers power consumption [5]. When the joints are very stiff, timing errors and irregularities in the floor can cause disastrous backlash, easily causing the robots to fall. Our best gaits had very low stiffness for the ankles, and medium stiffness in the hips and knees. Cycling this for the support and swinging leg also helped to reduce the backlash.

Another crucial tweak, which is also employed by Aldebaran, is to artificially compensate for weak hip motors by exaggerating the angles for hip roll. This constant often needs to be tweaked per robot, and occasionally for each hip joint individually.

To help maintain the robot's balance, we also introduced an additional compensator based on the angleX, angleY computed by Aldebaran on the micro controller in the chest. By measuring the tilt of the body, we were able to carefully compensate by increasing or decreasing the lean of the body.

On the implementation side, these tweaks were mostly synced directly with the gait cycle. For example, the angle of the foot relative to the ground followed a quarter period of sinusoidal motion during the swinging phase, but stayed constant during the supporting phase. Some constant, like the static forward lean, are applied regardless of the gait cycle.

4.5 Motion Metrics

Although we did not have the fastest walk, our final gait was extraordinarily stable. Coupled with our ability to dynamically switch velocities, we consistently beat even the faster teams to the ball. In addition, we fell over only once during the whole tournament, even when colliding with opponents. Our top speed at RoboCup was measured in the 10-11 cm/s range. However, with some more attention to sensor feedback, this could be increased to at least 15 cm/s while still maintaining responsiveness.

5 Behaviors

Automaton (FSA) system we developed in 2008, our goal this year was to advance our behaviors to a competent and consistent soccer player. We found our Python FSA system to permit rapid

behavior development and ease our work flow. Adding in new modules and extending our behaviors was simple and pain-free.

5.1 Finite State Automaton

For our behaviors we used the same FSA-based system that we developed in 2008. This FSA system keeps our code modular and organized. The FSA uses an API which clearly states how and when states within the FSA are exited and entered with clear links between each state. Our behaviors are written in Python, like our Aibo behaviors, chosen because it has a very expressive, simple syntax and is an interpreted language. As an interpreted language, Python does not need to be recompiled after changes. This allows us to reload modules individually without recompiling the underlying C++ structure. Practically, this gives us the ability to reinstall Python behavior files onto the robot and, without shutting down NaoQi or our other systems, load the new behaviors. Since restarting NaoQi can take a few minutes, this has saved us a lot of time during development.

We use an abstract FSA class to give our system its extensibility. This abstract class outlines the basic structure of an FSA and handles its execution. It holds the *run()* and *addStates()* functions, along with the state switching functions and other FSA helper functions. The *run()* function is called every vision frame by the behavior control module, the "Brain," and has the responsibility of executing the current state. Along with these functions the FSA class maintains a timer and frame counter for the current state, a reference to the last state run, and a reference to the last different state run.

Each FSA is defined by a class file that extends the abstract FSA class and by any number of state files. The class file is responsible for initializing the FSA and collecting the appropriate states into the FSA. It calls the FSA *addStates()* function to create a listing of all the possible states for that behavior. The FSA *addStates()* function uses Python's built-in *dir()* function to put all the states into a states list for later access. Each state is a function, taking only the behavior object, such as the player, as a parameter. Program 1 shows the implementation of a basic state.

Program 1 A basic Python state implementation

```
def spinLeft(player):
    if player.firstFrame():
        turn = motion.WalkTurn(150.,30)
        player.brain.motion.setNextWalkCommand(turn)

    elif (player.shouldWalkForward()):
        return player.goLater('walkForward')

    return player.stay()
```

The FSA API requires that each state return one of three possible values, given by three FSA functions: *goNow()*, *goLater()*, and *stay()*. *goNow()* switches and runs the new state immediately, without waiting for the current frame to end. *goLater()* switches states so that the new state is run in the next frame. *stay()* does not alter the FSA's state. There is another function, *switchTo(newState)*, which is used to switch states from outside the FSA. The functions that define state switches, like *shouldWalkForward()*, are found in the FSA player file.

One main advantage of the FSA is its extensibility. It is very simple to add another FSA to the Brain module. The Brain only needs to construct the FSA and call its *run()* method once a behavior cycle. In 2009, we currently use four concurrent FSAs. This allows us to break down larger behavioral problems and divide responsibilities in a manageable way. Three of our FSAs are directly behavioral components: a general player behavior, a head tracking controller (the "HeadTracker"), and a navigation controller (the "Navigator"). The fourth FSA is the fall controller and deals with the behavioral interruptions that occur in the special case of the robot falling over.

5.1.1 Player

The "player" behavior is the high level soccer behavior. It controls the HeadTracker and the Navigator, executing soccer strategies. According to the strategy and game role set by the playbook, the player directs the robots behavior. The player was broken down into many groups of states: ball chasing, ball finding, kicking, position states, goalie states, penalty kick states, and game states, each with their own state file. This further modularizes the system and lays out the general soccer playing strategy

CHASER State files used for the CHASER role.

ChaseBallStates States for the CHASER role, used for approaching the ball.

chase The state switched to when it is decided that the robot should chase. It returns a *goNow*, switching the player immediately to the appropriate state. Checks whether the robot needs to find the ball (in case it was lost between states), if and how it should approach the ball, and whether it should kick.

chaseAfterKick Used to chase the ball after a kick is made. Since the ball is often lost after kicking, due to a slow frame rate and a high ball velocity, the state assumes the kick went well, and turns the robot in the correct direction according to the last kick made. Also makes the robot track the ball.

turnToBall When the ball is at a great bearing from the robot, the robot will turn to face the ball.

approachBall The basic ball chasing behavior. If the ball is mostly in front of the robot, it runs directly for the ball using the omnidirectional gait. Used in situations when localization is poor and we are trying to get to the ball as fast as possible.

approachBallWithLoc For situations when localization is certain, instead of running directly to the ball, this uses the Navigator to position itself in a way to give it the best shot on goal. When approaching the ball from its own side, the player will line up with the ball in front of it, in line with the goal. When approaching from the opponent's side, it will position itself so that a side kick will be a shot on goal.

positionForKick This is used when the ball is close to the player and in front of it. Its goal is to put the player close enough to the ball, so that it may kick. When the player is close to the ball, it must step slower and with smaller steps than when it sprints across the field. Otherwise, it will kick the ball. **positionForKick** uses a gait that can takes smaller, shorter steps, and moves the player at a slower pace. In this state, the player does not use the full omnidirectional walk, instead moves only orthogonally.

dribble Using the dribble gait (a sort of duck stance), the player will walk through the ball towards the goal. As long as the ball stays in front of the player and it is still walking towards the opponent goal, it will continue to dribble.

waitBeforeKick Because the player can sometimes kick the ball when stepping close to it, it will wait after stopping to make sure that the ball is settled before it switches to kicking.

avoidObstacle When the player is chasing the ball, sometimes another robot or a goalpost can obstruct the player's path. Using sonar, the obstacle can be detected. If the obstacle is close to the player ($<\sim$50cm), the player will step around the object and track the ball with its head. If the ball's distance is less than 50cm, the player will not avoid the object and will instead continue chasing the ball. If a ball is this close, we want to be aggressive in chasing the ball and not risk giving up a kick.

ballInMyBox When chasing a ball in our own box, we want to stay trained on the ball, always facing it, but we do not want to walk into the box, since this is a penalty.

orbitBeforeKick When we approach the ball and we are facing our own endline, we do not want to kick the ball towards our own goal. If we approach close to the ball, and are facing the wrong the way, we will try to sidestep around the ball so that we are facing the opponent's goal.

FindBallStates States for the CHASER role, used for finding a lost ball.

scanFindBall When a ball is first lost, instead of spinning immediately, the player will scan for it in front of us.

spinFindBall If the ball is not found in front of us, the player will spin towards where the ball model says the ball is.

walkToBallLocPos If we do not see the ball, we start walking towards its location in our global ball model. This situation could arise if the ball is too far away to be seen.

KickingStates Houses all of the states necessary to kick the ball, including deciding which kick to make, side stepping so the ball is in the correct place for the chosen kick, and executing the kick.

getKickInfo Our localization is not always reliable enough for kicking, when the stakes are very high (scoring an own goal). In order to overcome this uncertainty, once the player has reached the ball, it will stop and look up, scanning from side to side. During the scan, the player averages the locations of seen goalposts as a quick-and-dirty localization method.

decideKick Using the information gathered during *getKickInfo*, the player will decided upon a "kick objective." This is a high level goal for the shot placement. The potential objectives are `OBJECTIVE_CLEAR`, `OBJECTIVE_CENTER`, `OBJECTIVE_SHOOT_FAR`, `OBJECTIVE_SHOOT_FAR`, `OBJECTIVE_KICKOFF`, and `OBJECTIVE_UNCLEAR`. The objective chosen depends on the player's orientation to both the opponent goal and his own goal, as viewed during *getKickInfo*. The player then switches to the appropriate kicking state for the chosen objective.

clearBall When the player is on his own side of the field, its objective is to get the ball off his own half as quickly as possible. Thus, it will use a strong kick and it is less

picky about its precise orientation on the goal. The player will use the kick which will send the ball deepest into the opponent side.

shootBall Uses the *getKickInfo* information to decide how to shoot. Chooses a kick based on location on the field and heading relative to goal posts seen. If there were not enough goalposts seen to make a good judgment, the player will, according to its chosen kick objective, switch to *shootBallClose* or *shootBallFar*.

shootBallClose The closer the player is to the ball, the more important it is that he is lined up correctly for the shot. The player will adjust alignment using the *alignOnBallStraightKick* state. If it is aiming within the posts, it will shoot, otherwise it aims for the center of the goal to minimize the chance of missing wide. Only uses localization information.

shootBallFar Instead of aiming at the center of the goal, the player will align so that it is shooting across the goal. This is so that from far away the ball will go around a goalie in the center of the net and lessen the risk of hitting a post on a crossing shot. Only uses localization information.

penaltyKickBall During a penalty kick, the player will align to kick the ball at the center of the goal. A simple strategy, but designed for the greatest chance of not missing the goal.

kickBallStraightShort, kickBallStraight, kickBallLeft, kickBallRight Execute the kick appropriate to their state name and aligns the robot so that the ball is in the proper position, relative to the feet, for the chosen kick.

sideStepForSideKick Decides whether the player must side step before executing a side kick.

alignForSideKick The player will side step until the ball is between the feet, where the side kick used in 2008 performed best.

stepForRightFootKick, stepForLeftFootKick Steps the robot sideways until the ball is in the proper place in front of the appropriate foot for a straight kick. Places the ball directly in the middle of the foot in the y-direction.

alignOnBallStraightKick Uses the Navigator to use the ball to orbit the ball a small amount. This is used mostly to make the robot face the goal.

kickBallExecute The player will run the kick chosen by the preceding states.

afterKick The player responds appropriately to whichever kick it used. First, the head will look in the direction that the kick should have gone. Second, for a side kick, the player begins *spinFindBall* immediately, since to chase the ball it will have to turn to it. For a straight kick, since the ball should be directly in front of the robot, it goes to *scanFindBall*.

DEFENDER States used by a robot in the DEFENDER role.

PositionStates States for positioning the player on the field.

playbookPosition The player gets its assigned position from the playbook and goes to it. It uses the Navigator to move to the point. It first spins to the point, then walks towards it, and finally it will use an omnidirectional walk to end up at the correct location and heading.

atPosition When the player is at its goal position, it will stay put until its current location and heading are no longer its destination location and heading.

spinToBall The player will spin in place until it is facing the ball. Used for when a defender should not move, but needs to be facing the ball.

spinFindBallPosition Similar to the chaser *spinFindBall*, but the player will not chase once the ball is found.

relocalize When the player's localization becomes poor (high uncertainty), the player must accumulate sensor data to find itself on the field. To accomplish this, the player will spin and pan its head so that it sees as many objects as possible.

5.1.2 Tracking

The head tracking FSA controls all movements of the head, such as panning and object tracking. The encapsulation of this functionality assists greatly in generating behaviors not only for game play but also from testing and training algorithms. While the ability to switch between the tracking of the ball and searching for landmarks from which to localize is extremely useful during games, the tracking of posts can be used to measure distance traveled, which will assist in such task as optimizing walking parameters or testing odometry or vision based distance estimation accuracy.

With the `HeadTracker`, The main player behavior is freed from the details of head movements and can instead issue high-level commands, like *trackBall*, *startScan*, and *locPans* (run localization pans). The `HeadTracker` deals with the details of where to set the head and implements intelligent tracking and panning behavior.

The `HeadTracker` has two associated state files:

PanningStates States for scanning the field, mostly for the ball and to improve localization.

scanBall The head will scan back and forth until it sees the ball. It begins scanning for the ball according to the global ball model. If the ball is close, it begins panning with its head low, and the head starts higher for two other sections of ball distance.

scanning Repeatedly executes the `HeadTracker`'s current head scan.

locPans Continually pans the head back and forth with the head tilted back. This allows the Nao's camera to see across the field and increases the chance of seeing a goalpost.

look The player will look in the chosen direction and then return its head to the starting position. This was used to make a quick glance in one direction, for gathering object information.

TrackingStates States for tracking objects, e.g. the ball or a goalpost.

tracking This is the main state for the module. While the object to be tracked is on screen, the head will adjust to center the object in the image. When the object leaves the screen for a significant number of frames, the `HeadTracker` will return to its previous state, likely scanning for the ball.

activeTracking Often times in the SPL, the ball will be a great distance from a player, but in little danger of being immediately moved. In this case, staring only at the ball as the player crosses the field will cause significant declines in localization accuracy. To

overcome this, the player will occasionally pan its head left and right to try and boost his localization. While not panning, the player tracks the given object.

5.1.3 Navigation

Like the `HeadTracker`, the `Navigator` removes responsibilities from the main player behavior. Its concern is the movement of the robot around the field. Walking commands from Python are sent only through the `Navigator`. This keeps the responsibility in one place, and keeps the potentially large and expensive number of identical walk command calls to a minimum.

The Navigator's most complex responsibility is the *goTo*. Calling *goTo*(x,y) commands the Navigator to direct the Nao across the field to (x,y) using localization information. The `Navigator` handles the starting, stopping, turning, and speed decisions needed to walk to a specific field location.

NavStates The `Navigator`'s state file.

spinToWalkHeading The first state in a *goTo*. The player will turn in place until it is facing its destination (x,y).

walkStraightToPoint Once the robot is facing its destination during a *goTo*, it walks to that point. It does not walk only in the x-direction, but adjusts its heading with some strafing and spinning to keep itself on course. If it finds it is too off course, it will return to *spinToWalkHeading* and readjust its heading.

spinToFinalHeading Once the player reaches its destination (x,y), it spins to its final heading.

walking When the player calls *setWalk*, the Navigator will switch into this state. The state handles calling new walk commands and prevents identical walk commands from being sent from Python to C++, a slow operation.

orbitPointThruAngle Used for circling around an object. The `Navigator` uses an estimated dead-reckoning to side step and spin around a point in front of the robot for a preset angle.

5.2 Coordinated Behaviors

We describe here the system of coordinated behaviors developed for use with the Aibo. This year we adapted these behaviors for the Nao platform with no major changes to the system. The intention of our behavioral system has always been to facilitate a robust coordinated system to allow for high level team play. We use a decentralized, dynamic role switching system, which utilizes the concepts of **strategies**, **formations**, **roles**, and **sub-roles** to build a robust system of position, based soccer.

5.2.1 Strategies

For competition we had the following formations: *Ready*, *NoFieldPlayers*, *OneField*, *TwoField* and *sThreeField*.

Strategies are the first and thus most abstract level of our coordinated behavioral system. Strategies specify a general manner of play to be used. A strategy is theoretically a set of formations,

which all share some underlying trait. The majority of switching between strategies occurs as the number of active team members changes due to penalties. We could define a strategy **sDefensive** which would require that every formation within it had two defenders. The logical extensions of how to build additional strategies are fairly obvious; however, the decision making process for determining which strategies should be used is a much more complex problem and is a target of current research within our team.

5.2.2 Formations

For competition we had the following formations: *NoFieldPlayers*, *OneField*, *DefensiveTwoField*, *NeutralDefenseTwoField*, *ThreeField*, *OneGoalBox*, *TwoGoalBox*, *Kickoff*, *Ready*

Formations act as a layer above the role switching system with each one selecting from a different combination of roles to determine the most appropriate one for a robot. The number of roles to select from is equal to the number of non-penalized robots. The *Field* formations are used for regular game play and assign a role by first selecting the best available chaser and assigning the remaining roles by player number. Although agents use *Field* for the majority of the time, the first formation called will always be *Kickoff*.

The kickoff in any soccer game is the situations where the information will be most clearly known to all members of the team: position and velocity of the ball, relative position of the opponents (depending on which team is kicking off and which is defending), and near exact positions of all teammates. Thus prior to kickoff robot A will always setup in position to start playing defense and robot$B$ will be in the center ready to become the chaser. Once play begins all robots will maintain their roles for a specified amount of time, so that robot A will move immediately to his defensive position and robot B will chase the ball.

The robots will move out of this formation and into *Field* after a short amount of time or if something unexpected occurs (i.e. the ball goes out of bounds, a player is penalized, etc.), but it should be pointed out, that play in the *Kickoff* does not look any different then other points in the game, the formation simply reinforces the method of soccer we wish to be played against any poor data which could occur during the very familiar kickoff situation.

The *GoalBox* formations are put into action whenever the ball moves near the team's own goal box. The goalie then becomes responsible for controlling the ball and all field players stop chasing the ball and are assigned defensive positions outside the goal box, so that an 'illegal defender' penalty is not incurred. Due to the goalie's poor mobility when in a deep squat, the goalie will only chase if the ball is in close in front of him and not moving. While this can result in none of our robots chasing the ball, we consider the risk of standing up and exposing the goal much greater than a game stuck penalty. The roles here are prescribed as *DEFENDER*, *DEFENDER* with the sub-roles for the two defenders specifying that one should maintain a left deep back position, while the other plays a right deep back role.

5.2.3 Roles

For competition we had the following roles: *CHASER*, *DEFENDER*, *OFFENDER*, *GOALIE*

Roles define the fundamental activity a robot should be preforming from frame to frame and directly influence which behavioral states in the player's FSM are to be used. The four roles used at RoboCup 2009 were *CHASER*, *OFFENDER*, *DEFENDER*, and *GOALIE*. Primarily an

organizational tool they collect and switch between subroles that assign exact positions. This switching is generally dependent on the position of the ball.

5.2.4 Sub-Roles

For further specifying what function a robot should serve on the team, we define sub-roles beneath the standard roles. Sub-roles facilitate two separate necessities in our system. First they specify parameters which refine the objective of a robot at any given time; for the most part this takes the shape of dictating positions for players based upon the location of the ball.

The second task accomplished is that sub-roles allow multiple robots to have the same role without conflict between the two players. Thus we can have two defenders who do not fight to maintain the same position in front of the goal, but instead take positions which allow them to better utilize their knowledge that there is another defender assisting in the overall defensive objective.

5.2.5 Role Switching

The vast majority of play occurs inside of the *Field* formations. With that in mind, the role switching system is based almost entirely around the idea of having the roles of *CHASER*, *DEFENDER*, and *OFFENDER*) to be allocated. The role-switching during *Field* operates around our philosophy of having one, and only one, robot chase the ball at any given time, while allowing any robot, save the goalie, to become *CHASER*. The *CHASER* role is allocated to the robot determined to have the minimum *chase time*. We define *chase time* as the estimated time it will take a robot to get to the soccer ball, calculated primarily around odometry. We assume that the robot is unobstructed and not delayed in moving to the ball for the calculation of *chase time*. To reflect our overall aims of scoring a goal, we reduce the assumed *chase time* in special circumstances – such as an agent lined up behind the ball going towards the opponent's goal. Lowering the *chase time* improves the likelihood that an agent not closest to the ball, but still in the best position to score, will become *CHASER*. Each robot broadcasts its *chase time*, when communicating to teammates, allowing each robot to determine the chaser independently and thus denying the need for any negotiated decision making. We feel that a non-negotiated role switching system is absolutely necessary to uphold our philosophy of reaching the ball as fast as possible, waiting just five frames to receive confirmation to chase could often cause a robot to lose the ball to opponent at high levels of play. Information lag and system noise incurred from sensor data may cause the agents' divergent world models to bring about different conclusions as to which robot has the shortest *chase time*. To combat this error while maintaining our non-negotiated system, we use a tiebreaker, which draws its inspiration from real sports, where certain players "call off" others (i.e. a goalie can call off a defender). For our system we use player numbers (each robot has a unique number 1-4), such that in a tie-break situation the higher player number can call off a lower player number.

If Robot A believes it might be the fastest to the ball (it is within a small threshold ϵ of the minimum *chase time* for all the robots), it will start pursuing and calling the ball ("I got it!"). Robot A will continue to chase the ball until a higher ranked robot calls it off, and/or any lower ranked robots receiving Robot A's call discontinue chasing. Thus Robot A is committed to chasing the ball. It cannot stop chasing and calling the ball until its *chase time* is outside a larger threshold δ (that is $\delta > \epsilon$). This prevents hysteresis, where robots oscillate back and forth between roles due to small changes in the data. There is a third important threshold, λ, which ensures

that a robot should stop listening to a higher ranked teammate if its *chase time* is less than that teammate's *chase time* by the value of λ. This ensures that when there is a discrepancy between local information and communicated information the robot relies on its local information. When things rapidly change on the field, a robot must not wait for a message from the current *CHASER* with higher rank before it acts. In summary Robot A will chase the ball if:

1. $chasetime(A) - min(chasetime(A, B)) < \epsilon$ and no higher robot is calling off A, or
2. $chasetime(A) - min(chasetime(A, B)) < \delta$ and it was already chasing and no higher robot is calling off A, or
3. $chasetime(A) < chasetime(B) - \lambda$ where the Robot B is the higher ranked robot calling off A

Each threshold controls a feature of the robots' cooperation. Increasing ϵ makes it more likely that a robot who is farther from the ball will ultimately end up being *CHASER*, but makes it less likely that no robot will be *CHASER*. δ controls how willing robots are to switch roles. Increasing the δ value decreases how often the robots will switch roles, which can leave the wrong robot chasing the ball, but protects against robots oscillating back and forth about who should chase. λ controls how much a robot should rely on local information. Increasing λ makes it less likely two robots will chase the ball, but slows down reactions to a ball suddenly being closer to a non-*CHASER* robot.

After deciding on which robot should become the *CHASER*, the remaining field players must decide which robot is to become the *DEFENDER* and which should become the supporting *ATTACKER*. The decision making process for this issue is the same as in determining the *CHASER*, only the determining metric is *distance to own goal* instead of *chase time*. We structure the tie-breaking thresholds about defense to ensure there is always a *DEFENDER* when communicated data deteriorates.

With only two non-*GOALIE* robots playing for a team, a robot that determines it is not the *CHASER* is assigned to the remaining role in the formation.

6 Conclusion

RoboCup 2009 was a success for the Northern Bites, both on the field and as a personal year. We completed our migration of our code base from the Aibo platform to the Nao, reusing functional sections and improving our structure with lessons learned from the Aibo. As a team comprised of undergraduate students, our success came from the "fire" we brought to the team. The entire team was personally invested in the team and kept enthusiasm high. The team came together and worked furiously to develop the best RoboCup team we possibly could.

A key component of our success this year was our motion engine. Being completely homegrown, we could tune, tweak, and alter it to fit our needs, in any way we wanted. The walk engine was one of our teams strongest assets this year. It was not the fastest, in fact, it was only about 50% as fast as the quickest walk, but its omnidirectional abilities and its stability gave us more advantages than speed could have. Relying on Aldebaran's closed-source walk does not represent progress in robotics. We open-sourced our walk engine in the hopes that other teams will expand upon it and push the league farther forward. Having our own walk engine which we could control allowed us to optimize our leverage of our other modules, including vision and behaviors, to get a coherent soccer system.

We did well in 2009 also because we were sure to treat our robots with care and keep them healthy. Just as a human athlete must keep healthy in the off-season to train and improve, so too must a RoboCup team prevent robot breakage if it is to continue to develop and test its algorithms on a real robot. There are simulators, and they can be a great aid to rapid initial development, but we have found that there is no substitute for testing on a physical robot. It helped us to iron out some of our more complex behavioral problems and it was necessary for tuning behaviors.

It is clear there is still a significant amount of effort to be done to expose the full soccer ability of the NAO robot. Although we have seen some significant technical limitations stemming from the hardware and software design of an entertainment grade humanoid robot, this year has shown us that high level soccer play is possible. Unfortunately, the league suffers from a great divide between teams who are able to effectively play soccer, and those which are still struggling with the formidable challenges by the humanoid form factor. It is our sincere hope that this report, along with our publicly available source repository [1] will help teams jump over the initial engineering hurdle and accelerate the level of soccer play in the SPL.

A Source Code

Our source code is available online using the `git` code version control system. The most recent code, along with a list of release tags can be found at `http://www.github.com/northern-bites/nao-man`. To checkout the code directly on a Debian Linux system, the following commands are relevant. Further information can be found in the included README file.

```
> sudo apt-get install git-core
> git clone git://github.com/northern-bites/nao-man.git man
> git clone git://github.com/northern-bites/tool.git tool
```

This will create two directories, `man` and `tool` which contain the robot source code, and our visualization and debugging tools, respectively.

References

[1] Northern Bites. Northern bites source code release for use in the spl with the nao robot. `http://www.github.com/northern-bites`.

[2] Eric Chown, Jeremy Fishman, Johannes Strom, George Slavov, Tucker Hermans, Nicholas Dunn, Andrew Lawrence, John Morrison, and Elise Krob. The northern bites 2008 standard platform robot team. Technical report, 2008.

[3] Stefan Czarnetzki, Sören Kerner, and Oliver Urbann. Observer-based dynamic walking control for biped robots. *RoboCup Symposium*, 2009.

[4] Tucker Hermans. Localization in the robocup standard platform league, 2009.

[5] Jason Kulk and James Welsh. A low power walk for the NAO robot. Technical report, University of New South Wales, 2008.

[6] Johannes Strom, George Slavov, and Eric Chown. Omnidirectional walking using ZMP and preview control for the NAO humanoid robot. In *RoboCup Symposium*, July 2009.

[7] Johannes Heide Strom. Dynamically balanced omnidirectional humanoid robot locomotion. Bowdoin College, 2009.

[8] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2006.