Mi-PAL Team Griffith — Competition report

Shane Anderson[1], Kristopher Croaker[1] Jeremy Dockter[2], Vladimir Estivill-Castro[1], Joel Fenwick[1], Nathan Lovell[1], and Stuart Seymon[1]

[1] School of Computing and Information Technology
Griffith University, Nathan Campus, Australia
WWW home page: gucis.cit.gu.edu.au/%7Emi-pal
[2] School of Information Technology
Griffith University, Gold Coast Campus, Australia
WWW home page: gucis.cit.gu.edu.au/%7Emi-pal

Abstract. This report describes the technology and approaches taken by the Mi-PAL team representing Griffith University in RoboCup 2003. We participated as a new team in the Sony legged league using ERS-210A Aibo robots. We have our own technologies for the challenges in vision and location that emerge in this application domain. We also attempted several modes of communication between the robots for strategy and team play.

1 Introduction

Robotic soccer (and in particular the RoboCup legged league) poses interesting and challenging problems [6]. We see these problems in a sequence, and will organize our report along this sequence. First, there are the issues of image processing and computer vision. It is hard to believe soccer can be played effectively by Aibo robots without recognizing the ball through computer vision systems. The image processing issues are complicated by the Aibo platform. Namely, the camera on board of the ERS-240A is not a high resolution camera with sophisticated optical lenses. Another complication is that a robot on legs makes the camera waggle significantly much more that robots on wheels (as in the middle size) or robots with cameras off-board (like the small size league). From the stream of images in the form of arrays of pixels of color, the software must interpret these and recognize objects. Our techniques and tools for vision-based object recognition are presented in [3]. The next challenge is locomotion, the robots must walk and get to the ball reasonable fast; otherwise the opposition will continuously play the ball first. The third challenge is to find where we are. This is crucial information to choose the correct action when actually controlling the ball. Thus, localization is the next problem than needs a solution for effective play. We regard team communication as the fourth aspect that demands a satisfactory answer. Otherwise, all players attack the ball, all concur to a common position and teammates end up colliding and interfering with each other. The objective of all these preliminary issues is to eventually collect sufficient information for the robot to carry out behavior and actions that are suitable

for the situation. We believe the next point in the sequence is the organization of decision making into behaviors and the control of actions by an architecture of behaviors. How can all this be organized within one CPU? This is the point of a software architecture. The mapping into a Software Architecture that uses the Sony object model for the Aibo platform and the the OPEN-R libraries for programing is the topic of Section 5. It includes a discussion of how the element of information of previous sections are collected into a common white-board and then analyzed for the construction of actions and the development of behaviors and states.

In all of these aspects, we emphasize that robotic soccer represents a very dynamic environment where decision must be made quickly and previous plans may need to be rapidly discarded. Plans are also rapidly discarded because of the intervention or style of play of the adversary. Also, we believe plans can not be very elaborate because it seems rather difficult to model all aspects that link the simple actions, like following a ball to the ultimate goal of winning the soccer match.

In our experience, developing software for the RoboCup competition represent a challenging Software Engineering effort. It is challenging in the fact that complex integration issues have to be resolved and that the task is so large that it demands effective collaboration in the team. It is also peculiar in that the development cycle is very short, and continuous iterative refinement is required to improve from one match to the next; however, a rigorous testing methodology and a scientific evaluation of the alternatives is usually avoided because of time pressures. Thus, we have taken an approach by which we believe more code and elements are to be tested and developed with the assistance of off-board methods. We would like to expand into developing with higher level descriptive and logically based tools than the plain C++. Finally, the difficulties in tracing actual faults in the software and identifying the source of 'bugs' prompted us to develop our own analysis tools for OPEN-R on Aibo. These aspects will be described in the later sections of the report.

2 Locomotion

The motion module is responsible for anything that causes a movement in the robot, and in particular, for making it walk forward, backward or perform a kick. We re-used the CMPack'02 Source Code Release for OPEN-R SDK v1.0 Copyright (C) 2002 Multirobot Lab [Project Head: Manuela Veloso] School of Computer Science, Carnegie Mellon University. We performed some minor modifications so it would compile with our code under our architecture, but otherwise was untouched. The main modification resulted from the fact that we could run the code in an ERS-210 with almost no problems; however, in the ERS-210A strange things would happen. It seemed that the faster CPU in the newer model allowed the motion module to be interrupted (removed from CPU cycles) and then other objects on the robot would send commands to the motion module par-

tially overwriting some of its state variables. When the motion module resumed, it was in an inconsistent state that even caused the robot to crash.

We developed a monitor module for the CMU motion object. We also run the motion object as a separate object. Therefore, the file

```
MS/OPEN-R/MW/CONF/OBJECT.CFG
```

includes a line for where the executable resides, namely

```
/MS/OPEN-R/MW/OBJS/MOTION.BIN.
```

In OPEN-R, all communication with the hardware is through the predefined object `OVirtualComm`. This object collects information from sensors and relays commands to motors. So, the setting of inter-object communication in `MS/OPEN-R/MW/CONF/CONNECT.CFG` includes the lines

```
MotionObject.MoveJoint.OCommandVectorData.S OVirtualRobotComm.Effector.OCommandVectorData.O
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S MotionObject.SensorFrame.OSensorFrameVectorData.O
MotionObject.Speaker.OSoundVectorData.S OVirtualRobotAudioComm.Speaker.OSoundVectorData.O
GUObj.MotionControl.MotionCommand.S MotionObject.Control.MotionCommand.O
```

3 Localization

We have developed a customized localization module. While we have observed that many other teams in the league have adopted a Kalman filter [4, 2] or a Monte-Carlo method to perform localization, we preferred our ad-hoc approach for several reasons. First, most of the time, a robotic soccer player does not need to know exactly where it is, if it has the ball in sight, it may be chasing it (or possibly negotiating with a team-mate if tracking rather than chasing should be done). Of course more advanced play will demand localization even while chasing the ball. As a new team, we decided on the simpler approach. While tracking the ball, until we do not get to control the ball, we may not use CPU cycles in localization. Once in control of the ball, localization may be required, but we argue that not necessarily (a red dog in control of the ball with the blue goal on sight may just kick straight ahead without knowledge of its position; refer to Figure 1). Another aspect that favors our approach is that the Aibo soccer environment is a difficult environment for incorporating odometry (which is usually very much embedded in the most common Kalman filter methods). Aibo's precision for odometry on movements is low and even if it were more precise, pushes by other robots as well as penalties by referees make this approach difficult. Similarly, the boundaries introduce discontinuities in the classic Kalman filter that uses multi-variate normal distributions as the probabilistic model updated by measurements.

Thus, our localization module uses triangulation from landmarks (goals and beacons) to obtain a reasonably accurate estimate of its localization. Our process ad-hoc because we use a series of functions where a unique point may be determined from bearings and distance measurements to landmarks, and in some cases, only a circular arc may be the localization set. By localization set we mean the set of points that are possible solutions to the constraints imposed by the sightings of landmarks. Most of the time, a single image is insufficient to

determine a unique point and bearing (note that localization aims at providing a position in the field as x, y coordinates and a bearing ϕ for the orientation of the robot). Thus, a set of sightings from approximately the last 5 or 12 frames (about one 5th or half a second) is used to carry out the triangulation. Clearly, because of the time discrepancy and different head movements the triangulations would not result in a unique point. As a matter of fact, event a robot who is not moving at all produces different object sightings (in terms of distance measurements to the objects) in consecutive frames. Therefore discrepancies in the triangulation emerge. Some landmarks are clearer at a certain distance, while angles may have extreme values in other positions that accentuate numerical error. We develop a series of techniques based on using the median of a series of values. The median is a statistically robust estimator, and should be preferred over the mean in this type of noise environment. With very careful tuning of what landmarks to trust more than others and their impact on the localization we develop the first phase of our module. It is capable of positioning a robot in the field with an error of 10 to 15 cm. It became fundamental for placing our goal keeper in position and maintaining it there, as well as returning it home when it chased the ball for a clearance.

The second element of our localization was to refine the localization based on the line markings on the field. We expected that knowing within 10 to 15 cm our position will allow the robot to determine if it was inside or outside a goal-box and then would work from there to obtain more precision for localization. Unfortunately the implementation of this aspect was not completed (and our attempt at the challenge for localization without beacons was unsuccessful). Similarly, our players would easily commit illegal-defender offenses as goalie-charging offenses.

The good point about or localization is that it did enable roles for players. It was possible to define a goalie, a defender and two attackers (a right wing and left wing). It enabled play without crowding too much. It also enable the team to patrol of the field when the ball went out of sight. More on this will be said later in this report.

3.1 Basic play

Players can make reasonable decision without fully localizing. In particular, because we obtain reliable bearing and distance information to corner beacons when close to them, an approximation to the best action to perform when controlling a ball in a corner is possible. Similarly, if the ball is controlled when the opponent goal is sighted directly in-front, it is not necessary to localize to execute a straight kick. Figure 1 illustrates that from the opponents half or own half, we shall perform a forward kick if immediately after controlling the ball, the opponent's goals is recognized to be in-front with respect to our orientation. Of course, sometimes the opponents goal may be in-front but not aligned for a shot. Immediately kicking forward may miss a goal opportunity, but advances the ball (which is advantageous because of the end barriers). However, it may be worth while to investigate conditions to allow for optimum performance in

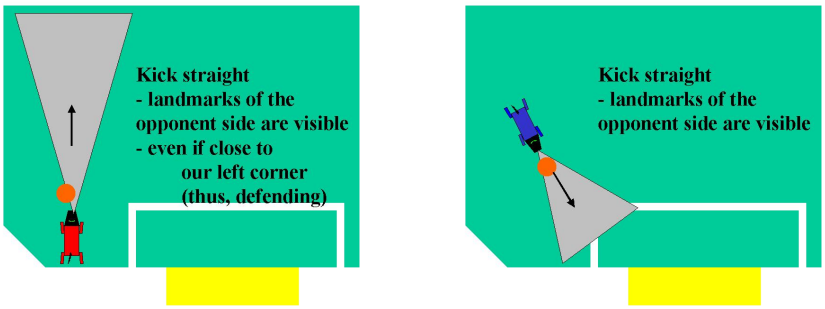

Fig. 1. When opponents goal is directly in front a straight kick can be performed without localization.

aligning a shot (while not been penalized for ball holding or challenged by the opposition).

3.2 Player roles

Our basic player is a 'penalty shooter'. It will search for the ball endlessly and when the ball is in sight it will chase it. If it manages to control the ball it will attempt to localize and then perform the best kick it considers appropriate. Searching for a ball not in sight means walking to the first of two home positions, performing a circular walk on the first home position and if the ball remains invisible, then walking to the second position and performing the circular walk there. We are unsure we have the best trade-offs for a circular search. While we found that a head sideways covered a large range of the field, the best speed to rotate is unclear. Too fast misses far away balls. To slow wastes to much time on a rotation. It also seemed difficult to break from the rotation into a walk towards the ball without loosing the ball again.

Another difficulty we have not resolved optimally is what to do after a shot. As a penalty kicker alone in the field, typically a head up will pick the ball again and the robot will continue the chase. However, in play the shot may be blocked and the ball is directly under the chin (but invisible for a head up). In this situation, the robot seems silly as it head home or does a rotation search when the ball is just right there. This situation also occurs when the ball is against a side edge and the kick leave the ball in the dogs chest (even if alone a in a penalty shut-out).

As we mentioned before, we do not need to determine precise position and orientation to decide on a useful kicks. Proximity to a corner is sufficient. This is illustrated by Figure 2 when attacking. One can assume that the corner beacon is in front, and then kicking to the opposite side is useful (i.e. if on right attacking corner, kick left). Of course this strategy is subsumed by the higher priority one that if the opponent goal is on site one kicks forward (in fact, our players

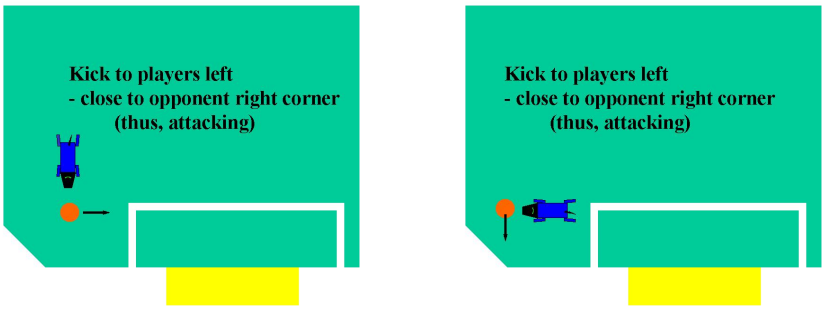

Fig. 2. When sufficiently close to an opponents corner, kicking to the opposite side is useful (i.e. if on right attacking corner, kick left).

also have a strategy that if the opponent goal visible on the left, they head-butt to the left). Similarly, there is a corresponding strategy for the corners when defending. This is obviously new to this year competition as the year before a defender in its own corner would have been penalized as an illegal defender. The

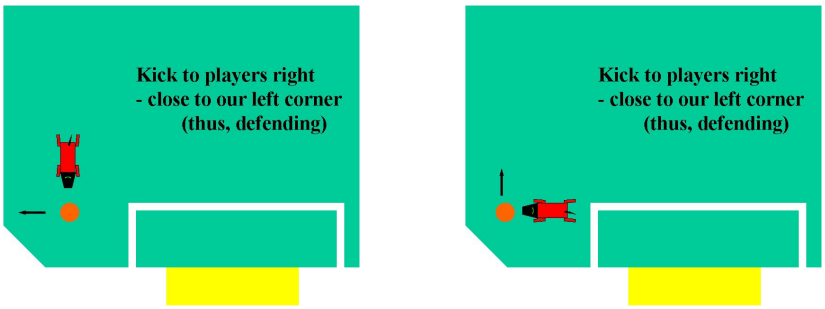

Fig. 3. When sufficiently close to our corner, kicking to the opposite side is useful (i.e. if on left defending corner, kick right).

strategy is than when sufficiently close to our corner, kicking to the opposite side is useful (i.e. if on left defending corner, kick right). Again, this rule is subsumed if we see objects on the on the opponents half that give an indication of a forward orientation. If facing forward while in our own right corner we kick left we actually center the ball for the opposition. Unfortunately, landmarks on the opponents half are far and may be blocked by robots in the field which makes reliable application of the high preference rule sometimes on reliable. However, most of the time defending robots end up chasing a ball against their own corner and thus do the correct action by this strategy.

Penalty kickers are ineffective as a team as they get into each others way. Localization allows to define zones for them to play and players assume roles by being penalty chasers while in the zones. This avoids some interference to team-mates, but is also a sub-optimal solution. We did not restrict our players to a zone, we expected that communication would eventually provide adequate compensation and a robot would broadcast control of the ball (slowing down the chase of others). Restricting players to a zone occasionally delivers the ball to the opposition. In particular, when the ball is in the zone of a penalized robot. Another scenario is that the ball may be in the opposite corner of a zone while closer to another team mate. What we did implement is two home points for each field player and a patrolling routine for when the ball is not on sight. The home points for the defender is the middle point of the top of its goal-box and the base point of the center circle. The first home points for wingers are the respective corner points of the attacking goal-box. The second home points of wingers are just the vertical intersection of the first home points with the half-line. We expected that broadcasting of sightings of the ball may assist breaking the team from unnecessary patrolling.

The goal keeper has only one home point. It has a "ball chaser" strategy embedded in it. Thus, while the ball is close enough it continues to chase and clear the ball. While this may lure the goalie totally out of its defending position all the way to the opponents goal-box it worked well for us for several reasons. An aggressive goal-keeper that attacks the ball as soon as it is in its proximity has a better chance to clear it and at least challenges the opposition for the ball. It can defend by prompting the opposition for a goalie-charge penalty. It usually clear the ball with a 5 second grab and a long kick, so if the kick is successful, the ball is far and thus it goes back to finding its home position. If the kick is unsuccessful it should immediately charge for it.

In fact, the our goalie is the player that exhibits more complex actions. It can track the ball with its head and walk sideways to align itself for a possible shut. It attempts to spread to a side if it identifies a shut coming in. If in control of the ball it will localize (the rules allow it to use a bit more time) and always clear to the side or slightly to the side (and not directly in front). It searches for the ball when the ball is not in sight differently (not rotating) but by head movements. Finally, it is the only player where we implemented decision-making based on odometry. Simply, a threshold was set experimentally so that when walking and moving long enough would cause a localization or a change of action.

4 Communication

We believe in a more flexible communication scheme based on occasionally broadcasting significantly mature information to team-mates. This is derived from the fact that we localize occasionally, therefore, little information can be communicated to team-mates on the status of the world. In particular, even the ball may not be easy to localize on the field if the robot who sights the ball has not localized recently.

But as we indicated before, a significant amount of mature information occurs rarely in our playing roles. This typically occurs when the player is in control of the ball and has completed an attempt to localize. It knows what kick it will perform. All of these information is useful to team-mates because an approximation to the location of the ball can be provided. Moreover, an approximation to the expected outcome of the ball can also be provided. The information is suitable for a broadcasting mode that does not need acknowledgment. The robot performing the kick does not need confirmation (acknowledgment) to carry out the kick it has decided to perform (rarely in soccer players communicate and exchange acknowledgments before a pass or kick is made).

We now provide illustrations of our communication strategy with respect to some of the kicking strategies mentioned before. Our first example is a shot on the opponents goal by our wingers (refer to Figure wingattack). Since wingers

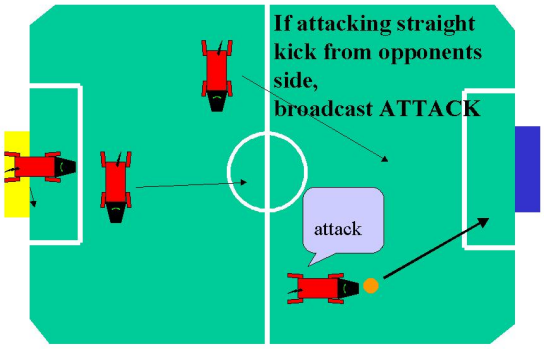

Fig. 4. A winger shooting on goal broadcast an attack message. Other players move to positions covering the edges of the region the ball is most likely to be after the kick.

are usually on the opponents half, such a kick will usually result on the ball on the opponents half. Our defender can go forward as well the symmetric winger who can center itself for a rebound from the goalie.

Similarly, a winger in the opponents corner performing a side kick indicates to the other teammates that the ball is on a quadrant of the field defined by the half line and the vertical line trough the middle. A *support* message is broadcasted and the other two field players cover the edges of this region (because of barriers these are the only two edges the ball can come out from the region and we have one player already in the region). Figure 5 illustrates this situation. In the German Open we found that, without these messages, opportunities to score where missed simply because the second winger was to far back while one winger had successfully driven the ball to a corner (never performing a kick, just pushing

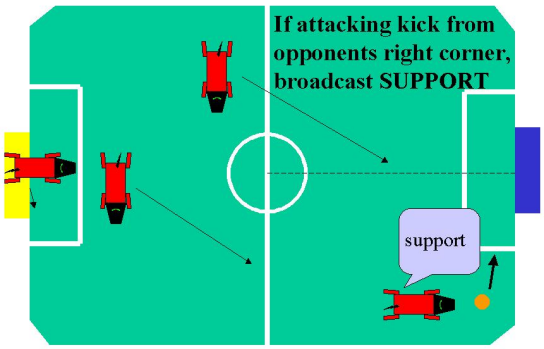

Fig. 5. A winger on an opponent corner calls for support and field team-mates cover the edges of the quadrant where the ball is.

it on the side barrier while attempting to grab it). The defender clearing a ball on out corner also broadcast a message we name *defend*. Figure 6 (a) illustrates this. We planed to include the goal keeper in this type of strategy. However,

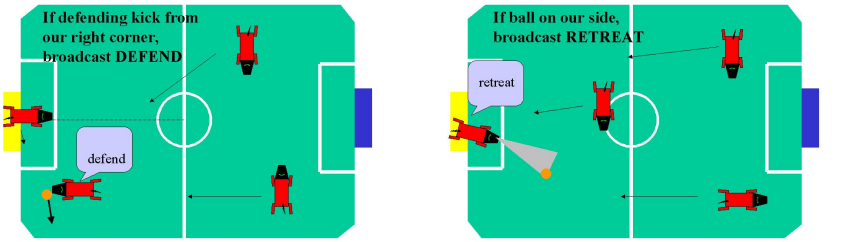

Fig. 6. (a) A field player on our corner corner calls defend and field team-mates cover the edges of the quadrant where the ball is. (b) Goalie calls retreat and field players come back and defend.

although our intention to participate document indicated we would be using dog to dog UDP and had this working, a few weeks before the competition we learned that this was disallowed. We switched to god-to-dog TCP and the final rules for wireless also disallowed this option. We were unable to implement the goalie's messages trough the TCP-gateway by the time of the competition. However, Figure 6 (b) illustrates the call from the goal keeper for the wingers to come to

the half line and be able to assist in contenting the ball after a clearance. Also, the defender retreats.

5 Architecture

In what follows we use Sony's terminology by which an object is *a separate process running under the Arperios operating system on the Aibo*. Such object is an instance of a subclass of the `OObject`[3] (refer to Figure 7) and must implement the methods `DoInit`, `DoStart`, `DoStop`, and `DoDestroy`. A series of important

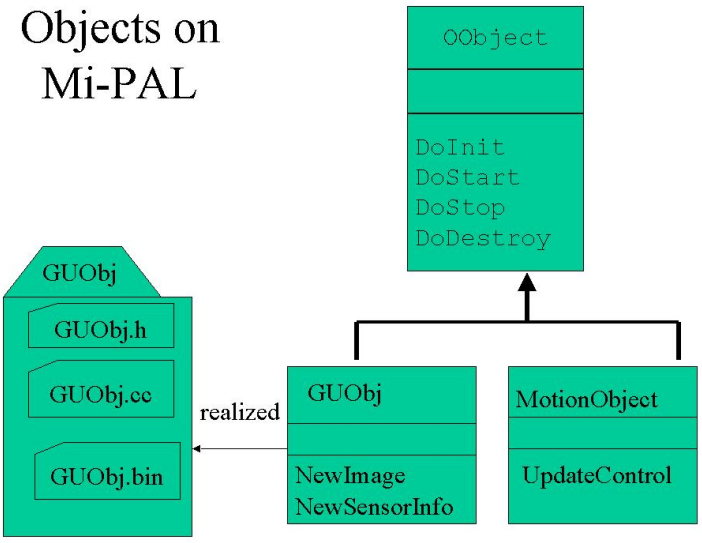

Fig. 7. UML class diagram to illustrate our high level architecture.

implications of Sony's object model on Aibo are as follows.

1. There can only be one object of each class.
2. Object communication can not be by arbitrary shared memory, but it most be arranged by interprocess communication that currently is described statically in the file `MS/OPEN-R/MW/CONF/CONNECT.CFG`.
3. Interprocess communication is controlled by the operating system and thus, it seems advantageous to reduce the number of objects. The following reasons justify such decision.
   - The opportunities for interprocess communication error are reduced if there is a less complex arrangement of interprocess links.

[3] SONY calls these "core classes'

– The interprocess delay/lag and context switches in the CPU are reduced if less process have to be managed by the CPU.

The object that wraps the Aibo's hardware is `OVirtualComm`. This object exists, and because we are using the CMU Motion module we require another object. Therefore, we aim for a software architecture that places all other functionality in just one more object [4].

The central object for our architecture is called `GUObj`. The main idea is that `OVirtualComm` will communicate to `GUObj` each time it has a frame (an image) ready. As a result, `GUObj` will execute the function `NewImage` and execute what we describe as a two phase cycle. The first phase is an analysis phase where the `GUObj` object collects information from vision (passes the new received image through the vision pipeline). This phase also includes collecting input from other sensors as well as messages from other robots or a PC (the game controller). This is done by querying components[5] in charge of these duties. The cycle speed is essentially driven by the reception of a new image and thus runs 25 times per second. The components in the analysis phase usually change information in what we call *the white-board*. This is a data structure were components have read privileges for all sections of the white-board, but have write privileges for only their section. This facilitates the communication between analyst components. Components can be organized as a pipe-line for example, or with other more flexible communication mechanisms[6]. The action components are the second phase of the cycle and these are typically of two types. The first type sends commands to the motion object while the second broadcasts information to other robots. This makes even clearer the need for a priority based semaphore on the motion object (recall Section 2). In our two-phase cycle, only one valid command (from an action component) is to be sent to the motion object.

Because of this, the file `MS/OPEN-R/MW/CONF/OBJECT.CFG` contains the line `/MS/OPEN-R/MW/OBJS/GUOBJ.BIN`. The lines that allow `OVirtualComm` to indicate that a frame is ready or that an event from a sensor has been received are in `MS/OPEN-R/MW/CONF/CONNECT.CFG` as follows (refer to Figure 8 for illustration of the static interprocess communication setting).

```
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S GUObj.Sensor.OSensorFrameVectorData.O
OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S GUObj.Video.OFbkImageVectorData.O
```

And the line that allows components in `GUObj` or other C++ action components to relay motion commands is

```
GUObj.MotionControl.MotionCommand.S MotionObject.Control.MotionCommand.O
```

[4] In fact this will not be possible, the robot to robot communication will demand the presence of other objects for this. In particular, the need to communicate trough a TCP-gateway as the rules for 2003 demanded and the need to link to the game controller placed other objects in our architecture.

[5] Note that we will reserve the term *component* for objects in the C++ sense, that is, instances of a C++ class, to differentiate them from SONY's objects.

[6] Note again that the data structure called white-board is our choice for C++ class communication as opposed to the inter-process communication for SONY objects.

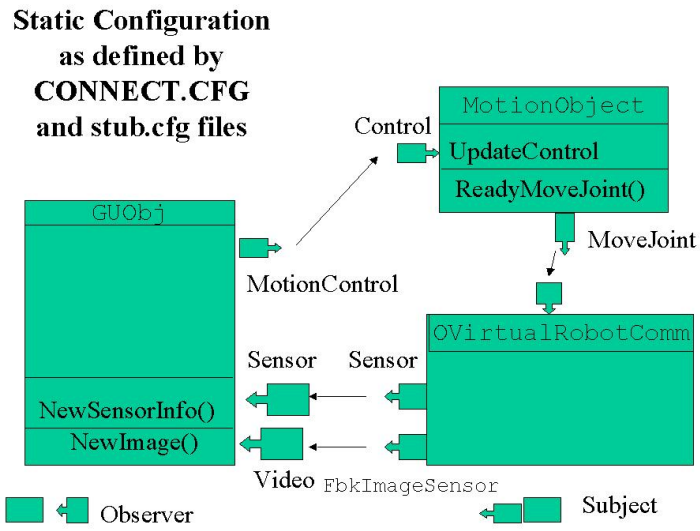

Fig. 8. Diagram to illustrate the inter-process communication settings.

6 States and Behavior

We defined a terminology for a subsumption architecture that controls the reactive play of the robots. The highest level is what we call *states*. These term has some similarities with the notion of *state* in finite state machines, or OMT/UML state diagrams [5, 1], but in our nomenclature has a less formal and more high-level semantics. A *state* is a general top-level, easy perceived, long lasting condition of the robot. Examples of states are *ready*, *playing*, *off-field*, *booting*, *set-team*, *getting-up*, and *returning-home*. We have a `state` component that is responsible for this meta-states or modes. Transitions between these states are usually driven by the touch sensors, but our state component includes a series of functions to carry out state transitions from code. This is used for handling the request from the game controller to move the robot from *playing* to *ready*.

This is what we mean by "easily perceived", the robot has direct indication that it must implement the state transition. Namely, its gyroscopes tell it is has fallen over, or it has received a signal (a push of a touch sensor) from *ready* (which means standing still) to move on to *playing*. The dynamics of the `NewImage()` function in GUObj is that a call to process the input by the state component happens before any *behavior* call. That way, *behaviors* can be implemented under the assumption that pre-conditions of the state are met. For example, a behavior may always assume the robot is standing (because if sideways the state would take over and perform the transition to get it up straight). This brings us to what a behavior is in our terminology. A *behavior* is a long lasting activity. Kind of a personality that defines what it the robot does in gen-

eral under this behavior and what it does if the personality is submitted to each of the possible *states*. Behaviors are constructed from actions or *commands* to the Motion Monitor. *Commands* are thus the detailed motions while *behaviors* are intermediate between *commands* and *states*.

The dynamics of the `NewImage()` function follows the call for the state to process the input by a call to the behavior to perform first its general step, and then calls for the behavior with respect to the states *playing*, *ready* and finally *returning home*. Each of this calls constructs a command with a priority, and therefore, the next call in the dynamics may overwrite the priority with a higher priority. The final step in `NewImage()` is to execute the *commands* constructed by the dynamics of state and behavior by forwarding them to the Motion Monitor.

We should point out that this allows a simple implementation by which the *state* component maintains if the robots is playing on the blue team or on the red team. For us, we could play both half of a match with the same memory stick. The robot goes through the states *booting*, to *team-toggle*, to *ready* with pushes in its touch sensors.

7 Tools

The challenge of developing software for the Aibo Robots this year had some differences from last year. For example, we did not have access to debug boxes any more. Some of this needs demanded the construction of some specific tools. We also had specific tools for our color calibration and the visualization a our location module. The first tool we describe is for developing the decision lists that segment an image. Our second tool runs a large portion of software in a platform that can run on the Aibo or on a PC. It allows the visualization of a pipeline of processes, from image segmentation, to blob formation, and then to object recognition. We also developed tools for localization that allow the visualization of the results of the localization results though and UDP connection to a robot in the field. Last but not least is our crash diagnostic tool. The need to operate the Aibo trough OPEN-R has advantages but has the disadvantages. The hardware is wrapped by an object, but disallows any possibility of an exception handler mechanism and when the robot crashes little information exists about the path of execution. Our tools allow us to recuperate a significant part of the stack trace.

7.1 Calibration tool

Figure 9 illustrates our tool for constructing a decision list for image segmentation. The tool is accompanied by a memory stick that allows to capture images on the field with an Aibo performing different task (the Aibo will capture images by walking back, swinging it head side to side, swinging the head down, or just straight ahead). The images are copied to the memory stick and then with the tool a human user can select portions and label them. The tool has an edge detection algorithm to facilitate selection of a region. However, when the Aibo

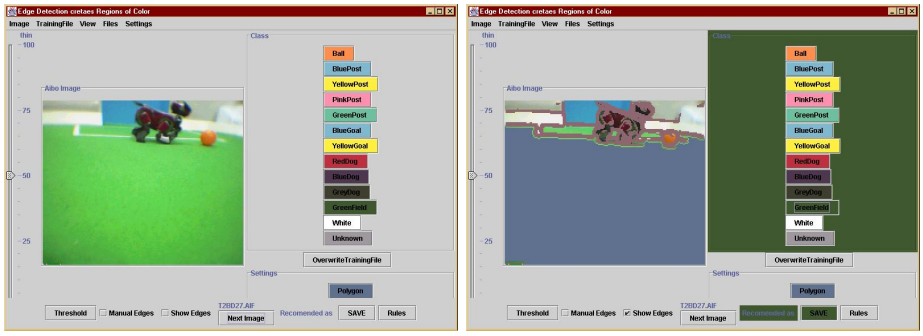

Fig. 9. A tools for creating a training set for building a decision list classifier.

moves the head fast blurred images have no edges and a square box is used to select a region. The training set is saved as a Weka training set and then the learning algorithm can be invoked from the tool. It was programed in Java and is platform independent. The decision list produced is then saved as a file that can be included in the memory stick of playing robots.

7.2 Vision Workshop

Figure 10 illustrates *Vision Workshop*. This tool verifies the a large proportion of the vision analysis code that is to run on an Aibo. Each component is presented as a window in the workshop and a data-flow can be indicated. In Figure 10, we see 4 components. Components can be enlarged and therefore we see two large and two small ones. In this figure, the first component can scan through directory of images, the second component shows the image segmented, the third component illustrates the result of forming blobs while the fourth component shows recognized objects. We have many other components, for example, there is a component for finding lines or edges in the image. It is also possible to use the tools to query the properties of the items displayed, for example, it is possible to investigate the coordinates of a blob center or the number of pixels that are in a ball.

7.3 Stack re-constructor

We develop two tools grouped in a package. The first we call `elp`. This tool is a perl script which parses `EMON.LOG` files. These are files that are generated when code on a Sony Aibo crashes (with suitable options when generating the memory stick). We recommend the use of this tool in conjunction with the second tool in the package named `StackEdit`. The first tools is similar to the EmonLogParser script supplied by Sony; however, in addition to obtaining the relevant information, `elp` goes on to call the Sony supplied program, *mipsel-linux-objdump*. The result of the parsing is assembly code. The assembly code generated by this call is written to an `aiboDis*.ass` file, and the code around the crash is printed

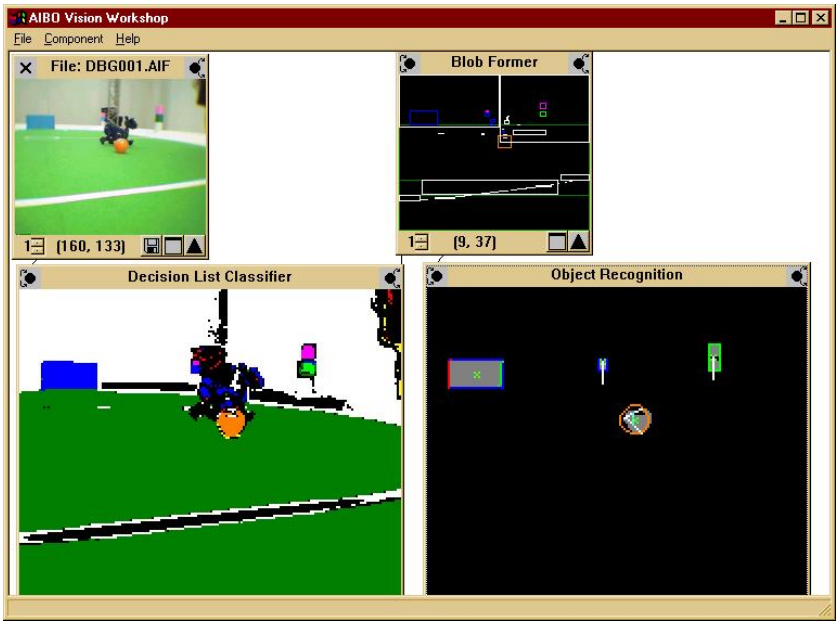

Fig. 10. A tool that visualized the vision pipeline as it analyzes an image.

to `stdout`. This is usually a large file written to the current working directory. We recommend to keep the resulting file where the relevant `.nosnap.elf` file resides (wherever the object that caused the problem was made), provided the offending object is known beforehand. However, `elp` will prompt users for the directory where the object was made if it is not the same as the `cwd`.

Our stack re-constructor is called `StackedIt` (although an earlier version was referred as `StackCheck`). It reveals which functions were called immediately before the crash. The accuracy of this trace depends on the method used to get the trace. The newer method is more accurate and we believe if does nor report any false positives unless it is told to. The older method n `StackCheck` is still available via command line arguments. Both methods make use of information from a disassembly of the `.nosnap.elf` of the offending object (thus the use of `elp`). The method in `StackEdit` starts with the function where the crash occurred. It proceeds to determine the stack usage of each function and finds the relative location of the return address. From this information, the method determines the previous function in the trace and the process repeats. Once this trace built is completed any values in the stack that could be return addresses can also be checked. These other checks will produce false positives in most circumstances so the output of these extra traces is clearly indicated and can be controlled with command line arguments. By contrast the older methods will output any values in the stack dump section of `EMON.LOG` which fall in the range

of addresses in the disassembly of `.nosnap.elf`. Any values that fit this range are mapped to the name of the function that contains that address. Since in some cases the return address is stored on the stack when a call is made we can work out which function called this one. This may produce some false positives, because not all values that appear on the stack are addresses. Thus, just because a value maps to a function, we can not guarantee that function is part of the *call trace*.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide*. Addison-Wesley Publishing Co., Reading, MA, 1999.
2. E. Brookner. *Tracking and Kalman filtering made easy*. John Wiley & Sons, NY, USA, 1998.
3. V. Estivill-Castro and Lovell. N. Improved object recognition - the robocup 4-legged league. In J. Liu, Y. Cheung, and H. Yin, editors, *Proceedings of the Fourth International Conference on Intelligent Data Engineering and Automated Learning, IDEAL-03*, pages 1123–1130, Hong Kong, 21-23 March 2003. Springer-Verlag Lecture Notes in Computer Science. Vol. 2690.
4. E.W. Kamen and J.K. Su. *Introduction to optimal estimation*. Springer, London ; New York, 1999.
5. J. et al Rumbaugh. *Object-oriented modeling and design*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
6. M. Veloso, W. Uther, M. Fujita, M. Asada, and H. Kitano. Playing soccer with legged robots. In *In Proceedings of IROS-98, Intelligent Robots and Systems Conference*, Victoria, Canada, October 1998.