Robocup Soccer Agent
Software Agents Final Project

Keith Holman* Jeremy Kuzub Amirali Salehi-Abari Karen Widish

April 1, 2008

Contents

*Authors' names appeared in alphabetical order.

1 Overview

The Soccer Simulation League division of RoboCup provides a software environment free of hardware design and development considerations. Teams compete on a server system in which individual players are instantiated as isolated execution processes. All process communication is through this server, which provides environmental 'sensory' information exchange to each player and performs requested actions. The challenge for team builders is in designing players which act together as a cohesive team with low-level tactics and high-level strategy.

Our agent team is based on autonomous agent principles, and meets this design challenge with a concept of "high-level" input and actions. Our goal in designing and implementing this software architecture is to provide a framework that encourages and supports this approach.

Our approach uses techniques well suited to these requirements. Sensory information provided to each player is processed using the JESS expert system library to infer high-level facts about the current tactical situation. For example, a player that sees a configuration of goal posts, teammates and opponents can infer the local offencive or defencive situation, and its own position within this context.

This high-level view of the game does not directly trigger player actions. It is used instead as input for a state machine implemented with the UniMod library. This allows the agent to act on past input sequences and effectively follow a strategy determined by the sequence of high-level inputs received from the Expert System. In practise, this provides the ability to assume an offencive or defensive stance and to react to subsequent events in that context. In order to prevent the state machine from becoming unwieldy, a hierarchical approach was taken.

This finite state machine issues outputs on transitions (Mealy model) which themselves are high-level actions. These are broken down into low-level actions executable by the soccer server. These action classes are deterministic and act as an abstraction layer between the server and the externally modifiable player behaviours. The high-level view of our agent model is presented in Figure 1.

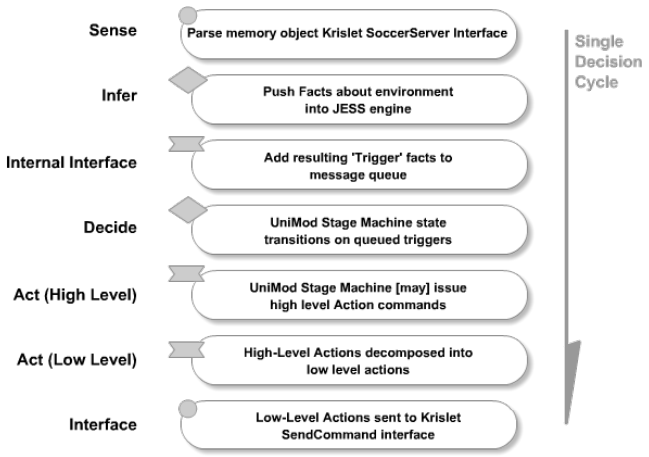

Figure 1: high level view of the agent model

2 Expert System using JESS

2.1 Introduction

In this application JESS functions as a forward-chaining engine. This section details JESS source code (CLP), which is the core of our induction engine in agent project. CLP code is similar to LISP in terms of syntax. The code consists of several rules (deffrule) and functions (deffunction) that let the engine extract high-level perception from low-level facts which are observed by an agent (soccer player). This high-level information will be used by the action selection component of the agent. As mentioned, we have selected Finite State Machine (FSM) to perform the action selection.

The explanation about the inputs (low-level facts) is presented in section 2.2. The outputs of our induction engine are detailed in section 2.3 and the challenges of extracting high-level information are described in section 2.4.

2.2 Inputs: Inserting Facts

The objects seen by the agent in each cycle should be inserted as facts to the induction engine. In the application presented this report, a specific template called "object" is defined in our CLP code to facilitate this insertion mechanism. Therefore, all observed[1] objects have to be inserted with this structure. The *object* template includes one multi-slot component (ourType) and two single-slot components (distance and direction). The structure of the template is presented in Definition 1.

The inserting object may have different types, which include different flags, players (teammate or opponent)

[1] as perceived from the soccer agent's perspective

Definition 1 Objeect Template

```
(deftemplate object
    (multislot ourType)
    (slot distance
        (type FLOAT)
        (default 0)
    )
    (slot direction
        (type FLOAT)
        (default 0)
    )
)
```

and ball. In case of players, the variable *ourType* is either *teamMate* or *opponent* along with a unique identifier. An example for *teamMate* and another for *opponent* follows:

(**object**(**ourType** teamMate 1) (**distance** 1.7) (**direction** 30.0))
(**object**(**ourType** opponent 3) (**distance** 9.3) (**direction** -15.0))

Inserting facts about the ball, flags and goals are similar, some examples are:
(**object** (**ourType** ball) (**distance** 10.7) (**direction** 0.0))
(**object** (**ourType** goal r) (**distance** 19.7) (**direction** -5.0))
(**object** (**ourType** flag g l t) (**distance** 47.7) (**direction** 7.0))

In addition to the *object* template, two additional simple facts should be inserted in the system for correct execution. The "*side-is x*" fact illustrates that the goal of the agent is in x side. x can be either l or r, which represent left and right respectively. Another fact, "*game-is on*", is inserted if the game is on.

2.3 Outputs

The following stimuli(events) are accessible through JESS, the definition of each in the case of availability is explained:

- *havePossession*: the agent is close enough to the ball that it is able to kick the ball.
- *seeBall*: the agent is able to see the ball in this cycle.
- *offense*: the agent is in the opponent's side of the field.
- *defense*: the agent is in its team's side of the field.
- *clearPass*: the agent believes it can successfully pass to at least one teammate without any interruptions.
- *clearShot*: the agent believes it can shoot toward the opponent's goal without any interruptions.
- *closestToBall*: the agent believes it is the closest agent to the ball.
- *teamMateClosestToBall*: one of the agent's teammates is closer to the ball.

2.4 Challenges with JESS

In this section discusses the challenges in extracting high-level outputs. In addition, this section describes implemented techniques and algorithm to overcome these challenges.

2.4.1 Offence? or Defence?

At first glimpse, figuring out if an agent is in an *offence* position or a *defence* position appears straightforward. The initial intuitive idea for determining this is analyzing the agent's position relative to the middle line; more specifically, an agent is considered in the offence position if it is passed the middle line, else it is in the defence position. However, this is not the case because the agent does not have a global perception of the field. Without this perception, the agent is unable to know its location within the field. This information is vital to the high-level decision making required of the agent. Therefore, an intriguing question is "how is an agent is able to determine its own location?" The approach presented uses "localization techniques" to solve this problem.

Using localization techniques, the agent should try to find its own location based on observed flags. Using simple geometry the agent is able to calculate the corresponding distances and directions from these flags allowing it triangulate its position. Since we only need to discern whether the agent is in offence or defence position and we do not need the precise position of the agent, which allows for a reduction in the complexity of the location calculations. Finding out if the agent is in the offence position is to some extent similar to finding out if the agent is in defence position. Thus, in the remainder of this report the explanation will be related to *offence*; applying these calculations to a (defence) position is trivial.

For the expert system to work correctly the rules to extract which side of the field is the opponent's side are based on the "*side-is x*" fact. For simplicity, the assumption is made for the remainder of this report that

the opponents are defending the goal on the right side of the field. If the agent has observed "*goal r*", "*flag g r t*" or "*flag g r b*"(see Figure 2) with the distance of less than 54, it should be in offence position. Figure 2 demonstrates the area that this rule can cover. The possibility exists that the agent is in offence position but it is not seeing any these flags because of the direction the agent is facing. To overcome this limitation the algorithm uses additional flags and associated rules.

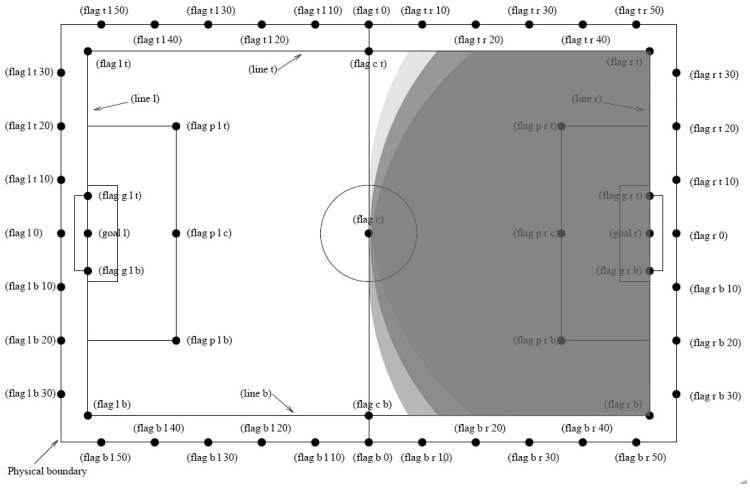

Figure 2: Coverage of the first Rule for offense position

The following rule is based on "*flag p r t*", "*flag p r c*" and "*flag p r b*". If the agent observes any of the mentioned flags and the distance of them is less than 34, then the agent concludes that it is in the offence position. Figure 3 illustrates the area that this rule can cover. The third rule

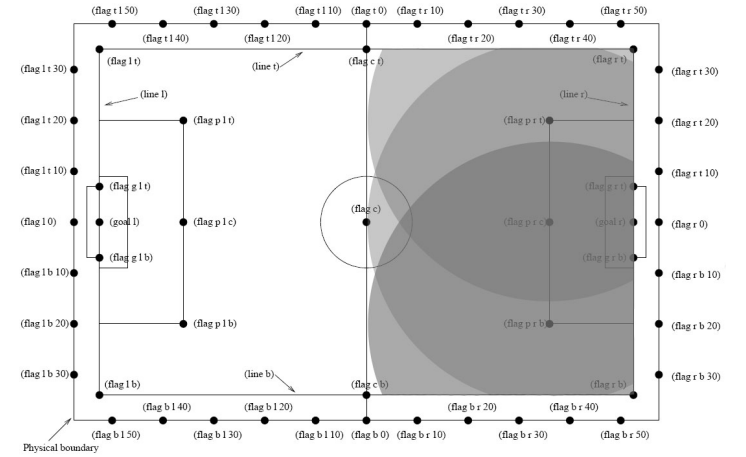

Figure 3: Coverage of the second Rule for offense position

to determine offence is related to the agent facing its own goal while it is located in the offence position. In this case, if the agent is seeing any flags around of its own goal and the distance of that flag is greater than half the field (50) and also the agent is able able to see the centre ("*flag c*").

2.4.2 Who is the closest to the ball?

Looking carefully to desired outputs of system presented in 2.3, there are two outputs (*closestToBall* and *teamMateClosestToBall*) which are related to the answer of this question, "who is the closest to the ball?" The distance between each observed and agent and the ball needs to be calculated. These values are calculated by converting the low-level data to a vector space model. Since an agent knows the distance and direction of all the objects it can observe, it becomes trivial math to construct a vector starting at the agent and finishing at the observed object. Figure 4 shows some examples of these vectors.

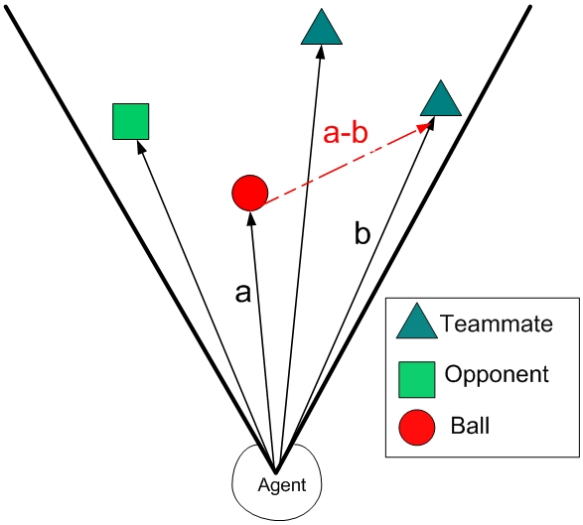

Figure 4: Vector Model Space

The difference between the vectors of observed objects are easily calculated since each vector has a reference point starting at the agent's position. As figure 4 demonstrates, knowing vector $\vec{a}$ and $\vec{b}$ for the ball and a teammate, respectively, the vector between the ball and teammate can calculated with the formula $\vec{a} - \vec{b}$. Consequently, the distance between the objects is equal to $\|\vec{a} - \vec{b}\|$.

2.4.3 ClearShot?

To understand whether an agent is in situation to have a clear shot, the agent should perceive three things:

1. The agent is able to observe the opponent's goal.

2. The goal is within a threshold distance; allowing the agent, considering its limited power, the ability to shoot the ball and it will reach the goal.

3. There is not any opponents in the path of the ball or surrounding the path to intercept the ball.

To determine whether these conditions hold, a triangle with a vertex at the agent's position and an *angle bisector* intersecting the opponent's goal is established, as illustrated in Figure 5. The agent believes it has a clear shot if no opponents exist within the triangle and the distance to the goal is less than the medium distance the agent is able to kick.
Figure 5 demonstrates a situation where the agent does not have a clear shot as opposed to Figure 6 where the agent has a clear shot.

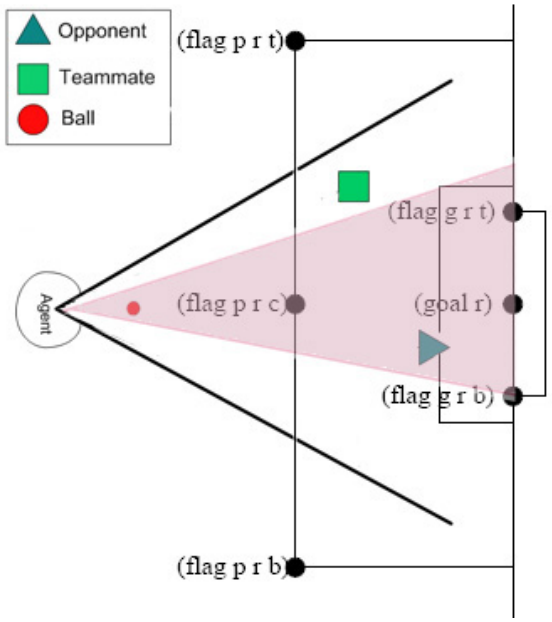

Figure 5: Demonstration of *clearshot* triangle while the agent does not have clear shot

This triangle method is preferred over other choices (for example, straight line, box, etc.) because a triangle model accounts for the fact that opponents further away from the agent's current position will have more time to intercept during the shot. Additionally, after the ball is kicked it will start to slow down giving opponents more time to invoke a defencive strategy.

The same strategy for *clearPass* with the only difference the implementation considers the positions of teammates as the target instead of the opponent's goal.

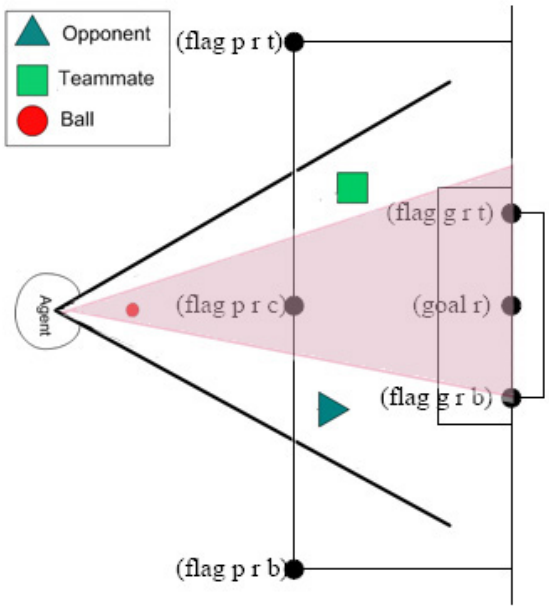

Figure 6: Demonstration of *clearshoot* triangle while the agent has clear shot

3 Finite State Machine using UniMod

The high-level knowledge returned by the expert system is passed into a finite state machine (FSM). This section describes the finite state machine design and implementation using the UniMod editor plug-in for Eclipse.

The highest level of the finite state machine is shown in Figure 7. UniMod requires, two special classes, an event provider and an object controller class. An event provider is a java class that generates the events that trigger the FSM. The object controller class is a class that handles the output and querying of the system by the finite state machine. These classes are specified first to UniMod.

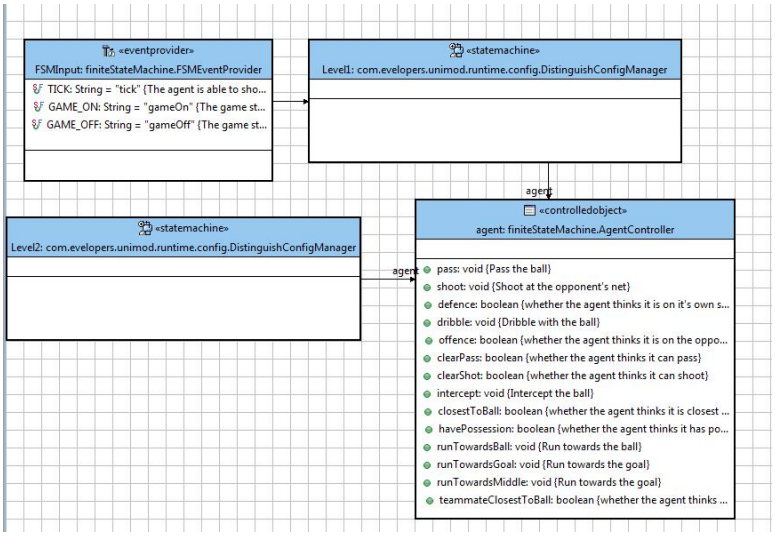

Figure 7: Finite State Machine Connectivity Diagram

The event provider sends three events to the state machine:

- *gameOn* - This event is sent when the game has started.

- *gameOff* - This event is sent to the agent when the game stops.
- *tick* - This event notifies the FSM a time interval has elapsed and an action may be returned.

As shown in Figure 7, the event provider sends the events to *Level1* of the state machine. *Level2* of the state machine is nested within *Level1*, as shown in Figure 8.

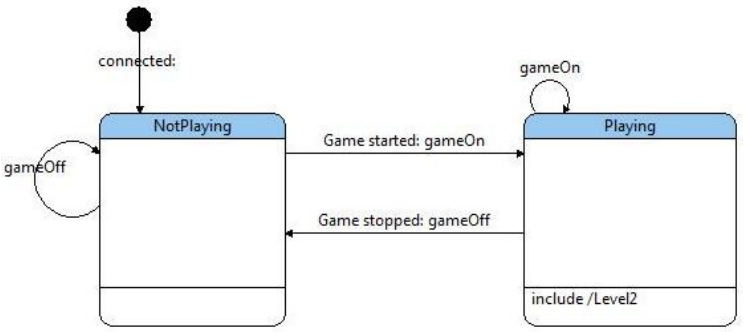

Figure 8: Level1 of FSM

Level1 of the FSM controls whether the game is in a playing mode or not. *Level2* contains more complicated FSM logic that translates high-level knowledge to high-level actions to play the game of soccer. *Level2* queries the current attributes of the agent with the controlled object and guard conditions, then a high-level action is called on the agent using the same controlled object, as shown in Figure 8. The FSM returns high-level actions. These actions are called high-level because they are a composition of simpler actions understood by the soccer server to complete the overall task. The agent keeps performing the action until the FSM changes state. The guard conditions are determined by querying the current attributes of the agent on each *tick* event. The attributes are set by the values returned from the JESS implementation discussed in Section 2. The descriptions of the high-level decisions used in JESS are described in Section 2.3.

The actions the agent can perform are:

- *runTowardsBall* - Agent runs in the direction of the ball.
- *intercept* - Agent moves to between the ball and their net.
- *shoot* - Agent kicks the ball at the opponent's net.
- *runTowardsGoal* - Agent runs in the direction of the opponent's goal.
- *runTowardsMiddle* - Agent runs to the centre of the field.
- *dribble* - Agent moves the ball forward slightly towards the opponent's goal, moving to follow it.
- *pass* - Agent kicks the ball towards a teammate.

Figure 9 shows a simplified version of the finite state machine used in this project. This FSM is not a complete FSM, as not all states handle all events. The model within UniMod is more complex and syntactically correct. This simpler model allows for an easier understanding of the logic the agent possesses when playing soccer.

The basic idea of the design is the agent acts differently depending if it is in an offencive or defencive situation. When on offence the agent is more likely to shoot at the goal, while on defence it is more likely to pass to a teammate. In addition, on defence the agent may perform the intercept action, trying to get between the ball and the net to ensure the other team does not have an advantageous position to score.

4 Challenges with UniMod

This section discusses the challenges that we experienced during the implementation of the finite state machine using UniMod. In addition, this section describes the solutions that were used to overcome these limitations.

The largest technical hurdle in implementing a state machine model is "state explosion". In soccer there are many distinct situations a player may be at any point during the game. The initial modelling attempts become unwieldy. This is not unique to UniMod, but rather a design consideration for any FSM implementation. Conversely, oversimplification of the state machine can result in a purely reactive architecture, negating the benefits of a state-based agent.

Both these issues were effectively addressed by using a nested state architecture, in which state machines are embedded inside one another. This compartmentalizes design challenges, and simplifies implementation and debugging.

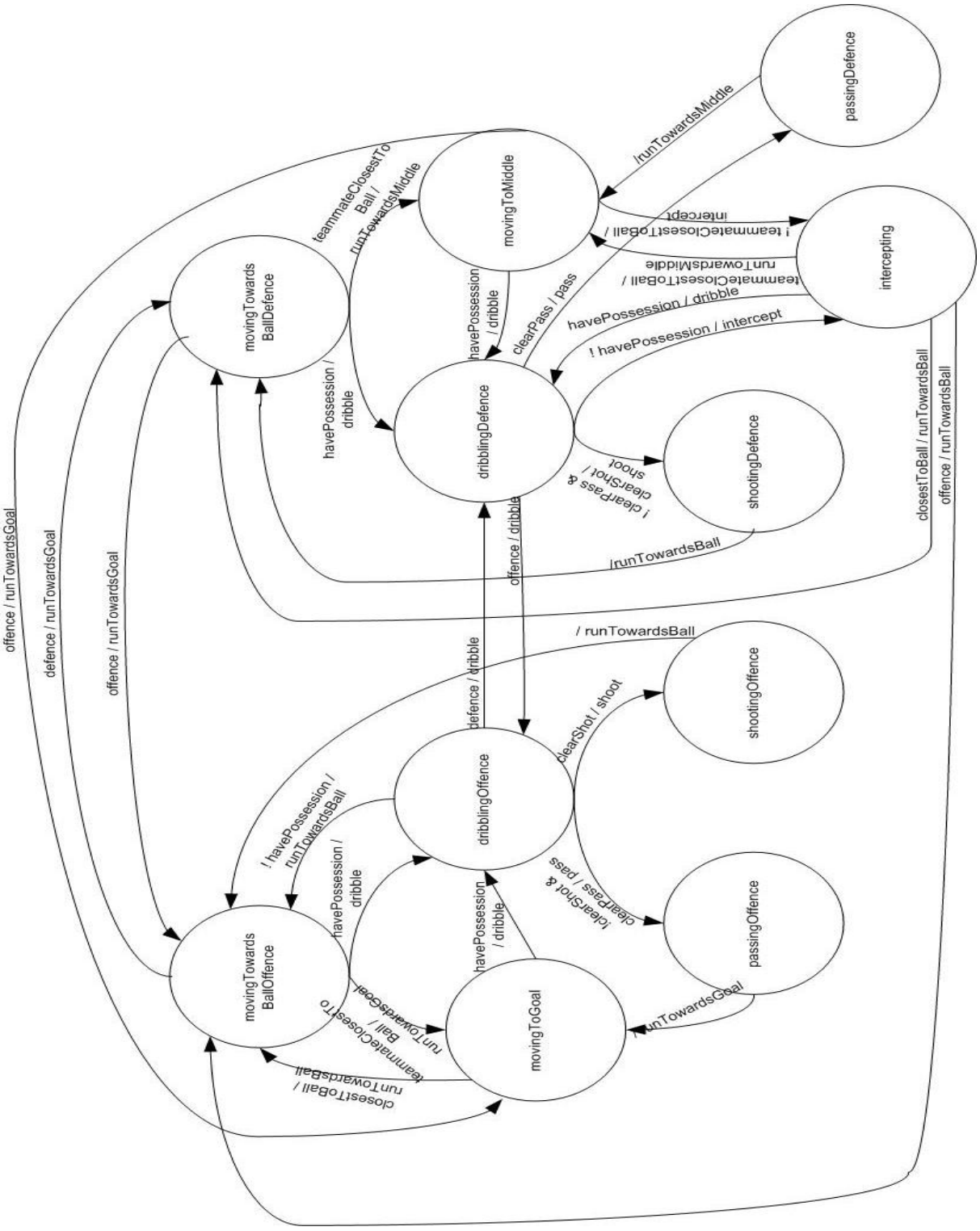

Figure 9: Level1 of FSM

5 Final Remarks

5.1 Library Integration

Our design requires the use of both a expert system and a state machine. Since neither of these could be realistically developed from first principles within one month, selecting pre-existing Java libraries became critical. JESS and UniMod were chosen for their ubiquity, support documentation, and established development history. Integrating these libraries with the soccer server communication protocols from the code-base became the main integration task.

In practise, documentation may be voluminous without being enlightening or accurate. A more useful source of information is obtained from online discussion boards with members who are currently integrating the libraries. Many of the errors, caveats, and development issues were a result of inaccurate or outdated documentation. Exchanging information with correct syntax between library interfaces is critical to functionality. In selecting two libraries to work serially, the number of unique interfaces to implement doubles.

To address this, library functionality was encapsulated within Java objects. A connection class effectively handles messaging passing between these encapsulated classes as well as the existing code base. This allowed for concurrent development without merging collision and compartmentalized development tasks.

5.2 Team Collaboration

Collaboration between team members during design definition and coding is key to the success of a delivery time constrained project. To aide in development the team leveraged existing proven project collaboration tools. The agent project code is maintained as an open-source project within the SourceForge community. This provides version control and online backup of the code base, as well as documentation support using a Wiki format. A hard drive failure during the development process had minimal impact because of the online storage practise.

Agent code was developed within the open source Eclipse Integrated Development Environment. Using integrated Subversion source control software connected to the SourceForge repository allowed smooth workflow between multiple developers. Eclipse specific plug-ins for both JESS and UniMod assisted in integration and design.

Project team members collaborated using the BaseCamp online project environment. This web-based application provides tools for maintaining online files, documents, discussion threads, and message broadcasts to team members. Milestones and action items are also tracked and updated as the development progresses. Using BaseCamp enhanced team coordination and minimized mixed messages and confusion.

5.3 Conclusion

Our results show that the *agent* approach creates distinct strategies and rich behaviours that are dependent on past sensory inputs. Execution using JESS and UniMod is acceptably fast and no modifications to timing were required. Using external text files to define the expert system rules and the definition of the state machine will allow further development of high-level behaviour.

Using online collaborative tools enhances parallel development at all stages of the project, demonstrating multi-agent systems benefit from effective agent communication and an established ontology.