SOCCER 2008
Simulation and Orchestrated Control of a Competitive Environment for Robots – Specification.

Dennis Peters[*] Software Design and Specification 9874

February 6, 2008

NOTE: All the following is subject to change.

1 Introduction and overview

A SOCCER system is made up of two kinds of components. A *simulator* simulates and displays the state of an imaginary game field. The simulator is also responsible for enforcing the rules of the SOCCER game. The moving objects on the field are players, divided into two teams, and one ball. Connected to the simulator are two *controllers*. Each controller directs the movements of one team of players.

The simulator and the controllers communicate over the Internet using the SCORE (Simulator-COntroler REquest) protocol described in this document. SCORE is layered on top of TCP/IP.

The remainder of this document describes:

- The rules of the SOCCER game.
- The details of the SCORE protocol.
- Other requirements on the components.

2 Rules of the SOCCER game.

The game being played has a passing similarity to real soccer, but is greatly simplified to make the project easier. The game described in this section is an ideal, being described in terms of real numbers. Furthermore, while it is based on the physics of the real world, it is not entirely realistic. For example we will ignore the third dimension of space and the angular momentum of bodies. Simulation introduces drift from this ideal and a later section will discuss the acceptable limits of that drift.

[*] The SOCCER 2008 project, and this specification, are based on SOCCER 2001 which is ©Theodore Norvell. It is used by permission.

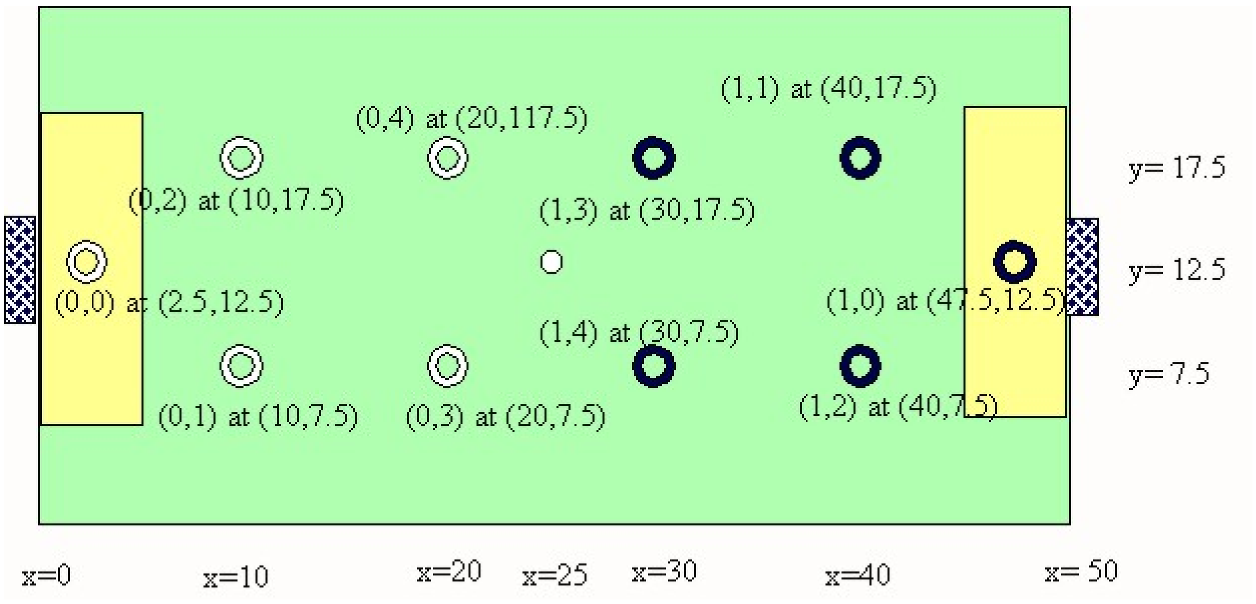

Figure 1: The initial positions of the players (not to scale). (Player (0,4) should be at (20,17.5).)

2.1 Field

1. A field is a rectangle with corners at (0m,0m) and (50m,25m).
2. Goals are 10 m wide and centred at either end.

2.2 Players

1. 2 teams of 5 players. Players are numbered $(0,0),(1,0),(0,1),(1,1),\ldots(0,4),(1,4)$.
2. Each player is cylinder of 25 cm radius with a vertical axis, and a mass of 10kg.
3. A player is either in play or out of play.
4. Aside from its number and the fact that it is in play, each in play player is fully described (at each instant) by the following state variables
    - Its position. (The position of the circle's centre.)
    - An orientation angle (its direction of motion) and a (linear) velocity.
    - Its angular velocity (turning rate).
    - Its (linear) acceleration.
5. Players may change their direction of motion with an angular velocity of up to 90 deg per sec in either direction.
6. Players may have a velocity of 0 to 5 m/s in the direction they point.

7. Players may accelerate by -5 to +1 m/s/s in the direction of motion.

8. The position of a player is not limited to the field of play. Unlike the ball, players may leave the field at any time.

9. Players of team 0 have a home goal on the left $x = 0m$ and players of team 1 have a home goal on the right $x = 50m$.

2.3 Ball

1. There is one ball – a cylinder of 15 cm radius, with a vertical axis, and a mass of 1 kg. It has a position and a velocity. For simplicity, the angular velocity of the ball is ignored, in our simplification of reality.

2. The ball will bounce off the sides of the field (except, of course at the goals) such that its velocity parallel to the side is unaffected, but the velocity normal to the side is half the original value. The side of the field is the line of reflection.

3. The ball will bounce off the goal posts. A bounce off a post is similar to a bounce off a side, except that the line of reflection is calculated a bit differently. The goal posts can be considered as points at (0,7.5), (0,17.5), (50,7.5), and (50,17.5). The line of reflection is perpendicular to a line drawn between the centre of the ball and the post. The velocity of the ball parallel to this line is unchanged. The velocity of the ball normal to the line of reflection is half its original value, and of course in the opposite direction.

4. The ball will bounce off players, with a coefficient of restitution of 0.5. There is no effect on the player's motion. See Appendix A for details.

5. The ball will decelerate due to friction at a rate of $v \times 0.1s^{-1}$

2.4 Kicks

1. Players within 100 cm (centre to centre) of the ball may change the velocity of the ball. The ball's new velocity vector is the sum of the old velocity vector and a vector that has magnitude from 0 to 10m/s and a direction that is equal to the direction between the player's centre and the ball's centre $\pm 45^o$.

2. A kick has no effect on the motion of the player.

2.5 Goals

1. When the centre of the ball crosses either goal mouth, one point is scored for the team whose home goal it is not.

2. After a goal the game resumes without delay, with the players that remain on the field in their initial states and the ball in its initial state.

2.6 Fouls

1. Players should not collide with players of *either* team. In the event of a collision.
   - If the point of collision is on the rear half of one player, the other player is charged with a foul.
   - Otherwise the player with the higher velocity is charged.
   - In the unlikely event of both players having equal velocity, either player may be charged.
2. Red carding: Any player that commits a foul is immediately removed from the field and is out of play for the remainder of the game.
3. The player that remains on the field after a foul is in no way affected. Its velocity is unchanged.

2.7 Reset

1. If for 30 seconds there has been no goal, no reset, and the ball has not been kicked by either team, bounced off any player, or bounced off any side or post, then the game is reset to the initial configuration.

2.8 Initial configuration.

At the start of each game and after every goal or reset the configuration is set so:

1. The ball is at a random position with $x = 25$ and $1 < y < 24$.
2. The players still in play are positioned as shown in the Figure 1.
3. All velocities are 0 in magnitude and orientations are 0^o for team 0 and 180^o for team 1.
4. All accelerations are 0 and no player is initially spinning.

2.9 Duration.

Each game will last 5 minutes.

3 Simulator Controller Request (SCORE) Protocol

***Note:** You will not be required to implement or interpret this protocol directly. The following details are provided to enhance your understanding of the project.*

3.1 Requests and replies

Each team controller may send requests to the simulator. The environment simulator will attempt to fairly merge requests from the two team controllers and to deal with and reply to each request promptly.

Each request from a team controller will be one line of text. For each request there will be a one line reply.

Each reply contains the time at which it was sent. This is useful because in any distributed application, there are small, but unavoidable delays, so it is useful to know when the information your application is receiving was valid.

In the event of a goal, a reset, or the end of a game, the next request from each team will not be honoured and does not get a normal reply. Instead of a normal reply, the environment simulator will send an indication that a goal was scored, that there was a reset, or that the game is at an end. See the grammar of the interaction for complete details.

Team controllers must not send a subsequent request before they receive a reply to the previous request.

3.1.1 Coordinate systems

The x direction goes from left to right, from 0 to 50m. The y direction goes from down to up, from 0 to 25m.

Positions are in meters, velocities in m/s, and accelerations in m/s/s.

Angles are all counter-clockwise from the x axis and are in degrees. Thus 90^o is up.

Team 0 will be attempting to score on the right and team 1 on the left.

Players are numbered from 0 to 4.

Times are in milliseconds since the start of the game. Thus the game ends at 300,000ms.

3.1.2 Requests and replies

The main body of the protocol consists of a sequence of requests, sent from team controllers to the environment simulator, and replies sent from the environment simulator to the team controller. There are two classes of request. "Feedback requests" request information about the game state. "Control requests" request a change in the state of some player. The simulator must reply to each request promptly.

Feedback requests In each case t is the time that (in milliseconds since the game start) at which the information is valid.

- `all?`

  Normal Reply: $t\ p_x\ p_y\ v\ \theta\ q_{(0,0)}\ p_{x,(0,0)}\ p_{y,(0,0)}\ v_{(0,0)}\ \theta_{(0,0)}\ q_{(0,1)}\ p_{x,(0,1)}\ p_{y,(0,1)}\ v_{(0,1)}\ \theta_{(0,1)}$
  $\cdots q_{(1.4)}\ p_{x,(1,4)}\ p_{y,(1,4)}\ v_{(1,4)}\ \theta_{(1,4)}$

  where (for all $i \in \{0,1\}$ and $j \in \{0,1,2,3,4\}$)

  - t is the current time in milliseconds since the game start.
  - (p_x, p_y) is the ball's current position in meters.

- (v, θ) is the ball's velocity in polar coordinates. We require that $0 \leq \theta < 360$.
- $q_{(i,j)}$ is 1 for players in play and 0 for red-carded players.
- $(p_{x,(i,j)}, p_{y,(i,j)})$ is the position of player (i, j) in meters.
- $(\theta_{(i,j)}, v_{(i,j)})$ is the velocity of player (i, j). We require that $0 \leq \theta_{(i,j)} < 360$.

Note:

- When $q_{(i,j)} = 0$ the values of $p_{x,(i,j)}, p_{y,(i,j)}, \theta_{(i,j)}$, and $v_{(i,j)}$ are arbitrary, although we still require $0 \leq \theta_{(i,j)} < 360$.
- Owing to simulation imperfections, the values reported may be slightly out of normal range, for example, a ball x position that is negative should not be considered an error.

Control Requests "Control requests" request that an action be executed either immediately or in the future. This is accomplished by a time field in the request. The simulator must act on the request at the earliest time possible which is equal to or greater than the time field's value. Thus, to request immediate action, a controller may use a time of 0. When a goal is scored, there is a reset, or the game ends, all pending actions must be dropped by the simulator.

It is *not* an error to send a request that applies to a player no longer in play (red-carded).

It is *not* an error to send a request with a time field greater than or equal to 300,000.

If, when a request is acted on, the player is no longer in play (red-carded), the request is simply ignored.

There is no communication from the simulator to the controller that the request has been acted on; the controller can assume that actions will be acted on at (or soon after) the time specified, provided the player involved is still in play and no goal has been scored in the mean time.

For all these requests, the normal reply is simply the current time in milliseconds since the game start. In all cases the player number must be in $\{0, 1, 2, 3, 4\}$.

- `spin!` *t number rate*

  number is a player number.

  rate is angular velocity in degrees per second (-90 to 90).

  Sets the angular velocity for the player.

- `accelerate!` *t number acceleration*

  number is a player number.

  acceleration is the rate of acceleration (-5 to +1).

  Sets the acceleration for the player.

  Note that while negative accelerations are allowed, negative velocities are not, deceleration continues uniformly until the velocity is 0. Similarly, velocities are capped at +5.

- `kick`! *t number velocity alpha*

  number is a player number.

  velocity is the change in the ball's velocity (0 to 10).

  alpha is the offset in degrees from a line connecting the player's centre and the ball's. (-45 to +45)

  If the ball and the player are too far apart, there is no effect. Otherwise, the new velocity vector for the ball will be new := old + velocity * u where u is a vector with magnitude 1 and angle alpha+theta where theta is the angle between the center of the player and the centre of the player and the centre of the ball.

  No player may kick the ball more than once every 0.5 of a second. It is fine to send a kick request, but the environment simulator will not change the ball's velocity, if the same player has successfully kicked the ball in the previous 0.5 of a second.

- `place_player`! *t number x y*

  number is a player number.

  (x, y) is the location to place the player.

  This command can only be used in "test mode", explained below. that the

- `place_ball`! *t x y*

  (x, y) is the location to place the ball.

  This command can only be used in "test mode", explained below.

- `set_time`! $t\ t'$

  The time is set to time t'.

  This command can only be used in "test mode", explained below.

3.2 Test Mode

In order to facilitate testing, the simulator must be capable of starting in either "game mode" or "test mode". The only difference between these two modes is that in test mode the `place_player`!, `place_ball`!, and `set_time!` commands are honoured.

3.3 Starting each game

The simulator must be capable of dealing with multiple games between the two team controllers. The motivation for allowing multiple games is largely to facilitate testing. Team controllers should offer to play at least one game, but may quit after the first game.

To get the ball rolling, so to speak, each team controller will identify itself and request a side. The environment simulator will send back a reply indicating which side the team will be. Team controllers should be capable of requesting either side based on user input.

3.4 Grammar for interactions

The following context free grammar shows the possible communication histories between the environment simulator and one team controller.

Neither the environment simulator nor the team controller should send more than 1000 bytes without a newline.

I use the following conventions

- The alphabet of this grammar is the set of 256 bytes tagged according to the direction they are going.
- *s2t*(E) indicates a communication sent from the environment simulator to the team controller.
- *t2s*(E) indicates a communication sent from the team controller to the environment simulator.
- This grammar does not show which replies normally go with which requests. That is done in section 3.1 on page 5.
- `typewriter font` indicates terminals. Each character is represented as a byte according to the ASCII code. This means that case (upper or lower) is significant.
- *space* indicates a byte of value 32_{10}, the ASCII code for a space.
- *newline* indicates a byte of value 10_{10}, the ASCII code for a newline (in Java and C/C++: `'\n'`).[1]
- The start symbol of the grammar is *Start*.

3.4.1 Common nonterminals:

$$\begin{aligned}
\mathit{Sps} &\rightarrow \{\mathit{space}\}_1 \\
\mathit{Name} &\rightarrow \{\mathit{Letter}\}_1^{10} \\
\mathit{Int} &\rightarrow \{\mathit{Digit}\}_1^{9} \\
\mathit{Float} &\rightarrow \mathit{OptMinus}\ \mathit{Int}\ .\ \mathit{Int} \\
\mathit{Letter} &\rightarrow [\mathtt{a}-\mathtt{z}] \mid [\mathtt{A}-\mathtt{Z}] \\
\mathit{Digit} &\rightarrow [\mathtt{0}-\mathtt{9}] \\
\mathit{OptMinus} &\rightarrow - \mid \epsilon
\end{aligned}$$

Note. No name should exceed 10 letters.

No Int should exceed 9 digits.

No Float should exceed 9 digits in its integral part, nor 9 digits for its fractional part. As you can see, the decimal point is *not* optional and must be preceded by and followed by at least one digit.

[1]Note that the java println subroutine may send both a byte 13 and a byte 10, and thus should be avoided.

3.4.2 Initialization, play and ending.

$$\begin{aligned}
\mathit{Start} &\rightarrow \mathit{t2s}(\texttt{SOCCER2008}\ \mathit{Sps\ Name\ newline})\ \mathit{s2t}(\texttt{SOCCER2008}\ \mathit{newline})\ \mathit{Init} \\
\mathit{Init} &\rightarrow \mathit{t2s}(\texttt{want\_side}\ \mathit{Sps\ Int\ newline})\ \mathit{s2t}(\texttt{on\_side}\ \mathit{Sps\ Int\ newline})\ \mathit{Play} \\
&\mid \mathit{t2s}(\texttt{want\_side}\ \mathit{Sps\ Int\ newline})\ \mathit{s2t}(\texttt{quit}\ \mathit{newline}) \\
&\mid \mathit{t2s}(\texttt{quit}\ \mathit{newline})\ \mathit{s2t}(\texttt{quit}\ \mathit{newline}) \\
\mathit{Play} &\rightarrow \mathit{t2s}(\mathit{Req})\ \mathit{Play}' \\
\mathit{Play}' &\rightarrow \mathit{s2t}(\mathit{Reply})\ \mathit{Play} \\
&\mid \mathit{s2t}(\mathit{GoalMsg})\ \mathit{Play} \\
&\mid \mathit{s2t}(\mathit{ResetMsg})\ \mathit{Play} \\
&\mid \mathit{s2t}(\mathit{EndMsg})\ \mathit{Init} \\
\mathit{GoalMsg} &\rightarrow \texttt{goal}\ \mathit{Sps\ Int\ Sps\ Int\ newline} \\
\mathit{ResetMsg} &\rightarrow \texttt{reset}\ \mathit{newline} \\
\mathit{EndMsg} &\rightarrow \texttt{end}\ \mathit{Sps\ Int\ newline}
\end{aligned}$$

Note that the *Name* parameter to the team's `SOCCER2008` message should be the name of the team controller in lower case. I.e., one of: red, green, blue, purple, orange, black.

Team controllers have the option of quitting before each game, if either team requests to quit then the simulator sends back a quit message and ends the connection. The three choices for nonterminal *Init* represent respectively the following three possibilities:

- that both teams request to play;
- this team requests to play, but the other requests to quit; and
- this team does not wish to play.

The Int parameter to `want_side` should be 0 or 1 and indicates which side the team controller wants to be. The parameter to `on_side` indicates which side the team controller really will be for the duration of the game. (If the team controllers want opposite sides, they get what they request.) The `on_side` message is not sent to either controller until both controllers have requested sides. The `on_side` messages signal the start of a game and thus are sent at time 0ms. No team should send a request before receiving its `on_side` message.

The parameters to `goal` should be the current score for team 0 and the current score for team 1 respectively.

The parameter to `end` should be 0 or 1 or 2. 0 indicates a victory for team 0; 1 a victory for team 1; and 2 a draw.

3.4.3 Request syntax

Request → `all`? *newline*
| `spin`! *Sps Int Sps Int Sps Float newline*
| `accelerate`! *Sps Int Sps Int Sps Float newline*
| `kick`! *Sps Int Sps Int Sps Float Sps Float newline*
| `place_player`! *Sps Int Sps Int Sps Float Sps Float newline*
| `place_ball`! *Sps Int Sps Int Sps Float Sps Float newline*
| `set_time`! *Sps Int Sps Int newline*
| *EndMsg*

The last kind of request should only be used if a team controller detects a problem with communications coming from the simulator. The simulator should end the game, when this happens. The team controller should print the erroneous message so that it is clear that the simulator did make a mistake.

3.4.4 Reply Syntax

The choice of reply format is entirely governed by the previous request.

$$\begin{aligned}
\textit{Reply} &\rightarrow \textit{Time newline}\\
&\mid \textit{Time Sps}\ \text{FourCoords}\ \textit{newline}\\
&\mid \textit{Time Sps Int Sps}\ \text{FourCoords}\ \textit{newline}\\
&\mid \textit{Time Sps}\ \ \text{FourCoords}\ \{\textit{Sps}\ \text{Int Sps FourCoords}\}_{10}^{10}\ \textit{newline}\\
\textit{Time} &\rightarrow \textit{Int}\\
\text{FourCoords} &\rightarrow \textit{Float Sps Float Sps Float Sps Float}
\end{aligned}$$

3.5 Error Handling

Any incorrect use of the protocol by either side must be reported by the other side. This includes

- any deviation from the grammar,
- numerical values that are out of range (but note that positions and velocities reported by the simulator may be a bit weird, these are not considered out of range)
- names that are out of range,
- attempts to use test mode commands when the simulator is not in test mode.

These errors should be reported by at the very least

- Printing an error message (to "standard output").

- Clearly printing the offending line (to “standard output”) exactly as it was received. For the simulator there must be an indication of the name of the offending client (red, blue, etc.).
- If possible, ending the game. Controllers send an EndMsg as the next request. Simulators send an EndMsg as the next reply to the next request from each controller.
- Optionally, simulators and controllers may pop up a dialog box to inform the user of the problem.

4 Requirements and Use Cases

4.1 Use cases for the Simulator and the Team Controller

Important note. *Normally Use Cases are phrased in terms of the “the system” and various actors that interact with the system. In our case, we have not a single system to be specified, but two kinds of systems: The simulators and the team controllers. Therefore these use cases are phrased in terms of the simulator, the team controllers, and various other actors.*

4.1.1 Top level Use Case “play game”:

Actors: The simulator, the two controllers (which I will call A and B), the user of the simulator (simulator user), the user of the A controller (A user), the user of the B controller (B user).

Initiated by the simulator user.

1. Simulator user: starts the simulator (see use case “start simulator” in game mode.
2. A user and B user: start the controllers (see use case “start controller”) using the host of the simulator and the appropriate port.
3. A and B initiate a handshake (see use case “handshake”)
4. A and B request a game (see use case “request a game”)
5. A and B complete the game (see use case “complete the game”)
6. A and/or B opt to quit (see use case “opt to quit”)
7. A, B, and the Simulator: close their ports and exit.

Alternative paths. The simulator should be prepared to play:

- 1 game (above sequence),
- 0 games (steps 1, 2, 3, 6, 7), or
- multiple games (steps 1, 2, 3, 4, 5, 4, 5, ..., 4, 5, 6, 7).

4.1.2 Top level Use Case “run tests”:

Actors: The simulator, a test process, a user.
Initiated by the user.

1. The user: starts the simulator (see use case “start simulator”) in “test” mode.
2. The user: starts the test process.
3. The test process connects to the simulators port twice.
4. The test process and the simulator proceed as in use case “play game” with the test process taking the part of both A and B.

Note: In test mode the simulator must deal with all requests.

4.1.3 Use case “start simulator”

Type: This is a partial use case.
Actors: The simulator, the user of the simulator.

1. User: instructs the OS to run the simulator.
2. Simulator: prompts for mode (game mode or test mode).
3. User: selects either game mode or test mode.
4. Simulator: Prompts for TCP port number.
5. User: enters the port number, or opts for default.
6. Simulator: Opens the port as a server.
7. Simulator: Listens on the port until two connections have been made to the port.
8. Simulator: Creates a window for displaying the game state.

Note: Step 8 can happen earlier or later, so long as the game state is displayed when the game is on.

Note: Although each Software Team is assigned a default port, each simulator should be capable of using any TCP port.

Exceptions: The simulator may fail to open the port. In this case it prints an error message to the console and exits.

4.1.4 Use case "start controller"

Type: This is a partial use case.

Actors: A controller, the user of the controller.

1. User: instructs the OS to run the controller.
2. Controller: Prompts for an Internet host address.
3. User: Supplies a host address.
4. Controller: Prompts for a port number.
5. User: Enters the port number.
6. Controller: Connect to the given port on the given host.

Note: In step 3, the user may be permitted to supply the IP host name, rather than the IP host address, but supplying the IP address must be an option.

Exceptions: The controller may fail to connect to the port. In this case it prints an error message to the console and exits.

4.1.5 Use case "handshake"

Type: This is a partial use case.

Actors, the simulator and the two controllers (A and B)

1. A: sends a `SOCCER2008` message with its name (colour) to the simulator.
2. Simulator sends a `SOCCER2008` message to A.
3. B: sends a `SOCCER2008` message with its name (colour) to the simulator.
4. Simulator: sends a `SOCCER2008` message to B.

Note, as long as 2 follows 1 and 4 follows 3, order of the above steps is arbitrary.

Exceptions: If a connection is lost, the agent should print a message, close its port(s) and exit.

If any agent detects an incorrectly formatted message, it should print a detailed error message (including the incorrect message and, for the simulator, an indication of where it came from), close its port(s) and exit.

4.1.6 Use case "request a game"

Type: This is a partial use case.

Actors, the simulator and the two controllers (A and B)

1. A: sends a `want_side` message to the simulator.
2. B: sends a `want_side` message to the simulator.
3. Simulator: initializes the game state and starts simulating.
4. Simulator: sends an `on_side` message to each controller.

Notes: Steps 1 and 2 can happen in either order. If A and B want different sides, they each get their request, otherwise the simulator selects one to be side 0 and the other to be side 1.

Exceptions: If a connection is lost, the agent should print a message, close its port(s) and exit.

If any agent detects an incorrectly formatted message, it should print a detailed error message (including the incorrect message and, for the simulator, an indication of where it came from), close its port(s) and exit.

4.1.7 Use case "complete the game"

Type: This is a partial use case.

Actors: the simulator and the two controllers (A and B)

1. One controller: sends a request to the simulator.
2. If the game has ended and the controller in question has not been informed yet

   - The simulator: sends an `end` message to the controller

   Otherwise, if there is a goal that the controller in question has not been informed of yet

   - The simulator: sends a `goal` message to the controller

   Otherwise

   - The simulator: sends an appropriate reply to the controller
3. Repeat from step 1, unless both teams have now been informed of the end of the game.

Note: Watch out. Once one side has been informed of the end of the game it may send a `quit` or `want_side` message. Ignore this message for now. On the other hand, when only one side has been informed of a goal, its further requests should be honoured.

Note: When both sides send requests at about the same time, the simulator must deal with them in some unbiased manner.

Exceptions: If a connection is lost, the agent should print a message, close its port(s), and exit.

If a controller detects an incorrectly formatted reply, `end` or `goal` message, or a reply with out of bounds values, it should print a detailed error message and send an `end` message to the simulator.

If the simulator detects an incorrectly formatted request, a request with out of range parameters, or, when in game mode, a request not allowed in game mode, it should print a detailed error message, and end the game, awarding victory to the other team.

4.1.8 Use Case "Opt to quit"

Type: This is a partial use case.

Actors: the simulator and the two controllers (A and B)

1. One controller sends a `quit` message.
2. The other controller sends a `quit` message.
3. The simulator sends a `quit` message to both controllers.

Alternative paths. One controller may send a `quit` message and the other a `want_side` message.

Exceptions: If a connection is lost, the agent should print a message, close its port(s) and exit.

4.2 Requirements for the Simulator

4.2.1 Functional requirements

1. The simulator must faithfully follow the rules of SOCCER described above.
2. The simulator must be capable of following the above use cases and must respect the SCORE protocol described above.
3. The simulator must report errors made by the controllers. The error report must contain the complete text of the erroneous message sent by the controller and the reason that the simulator decided it was erroneous.
4. The simulator must display the game state graphically as it progresses in real time, including the score and the positions of the players and the ball.

4.2.2 Performance

Performance requirements assume a platform consisting of Sun Java 1.5 on either Windows XP or (SUSE) Linux 2.6.5 on a Pentium IV 1.4GHz with 512 MB of memory.

1. The simulator must be capable of updating and redisplaying the game state at least 10 times per second.

2. The simulator must be capable of handling an average of at least 10 requests from each side per second.

3. Control requests with future times should be acted on no more than $1/10^{\text{th}}$ of a second after the requested time. Control requests with past (or present) times should be acted upon no more than $1/10^{\text{th}}$ of a second after they arrive at the simulator.

4. Control requests should not be acted on before the requested time.

5. The simulator must simulate the movements of players and the ball with a drift of no more than 25cm per second, unless there are bounces or other discrete changes to the velocity. In other words, no matter what happens, after 1 second every player and the ball must be within 25cm of its theoretically correct position; after 4 seconds it is 1 metre; and so forth. It is hard to quantify drift in the presence of bounces. For example the location of the ball after it has bounced off a player is highly dependant on exactly where the collision is simulated to take place. In the presence of bounces and kicks, the simulator should make a good effort at accurate simulation.

There is no specified upper limit on the processing rate, however I recommend care that numerical error does not creep in due to the use of very small time differences. You should also consider the effect of clock resolution on your simulation, i.e., avoid time steps of 0.

4.2.3 Nonfunctional Requirements for the Simulator

1. The simulator must be capable of running under either Windows XP or Linux.

2. The simulator must be written in Java.

4.3 Requirements for the Team Controller

4.3.1 Functional Requirements for the Team Controller

1. The controller must respect the use cases above and must follow the SCORE protocol.

2. The controller must report errors made by the simulator. The error report must contain the complete text of the erroneous message sent by the simulator and the reason that the controller decided it was erroneous.

4.3.2 Nonfunctional Requirements for the Team Controller

1. The controller must be capable of running under either Windows XP or Linux.
2. The controller must be written in Java.
3. The controller must be designed and written to make a reasonable effort to play the game well. For example a controller that simply spins its player for the entire game will be judged to be unacceptable.

A Calculating Ball Bounces

For simplicity we will assume that the ball bounces off players as if the player had infinite mass. In effect this means that there is no effect on the motion of the player when a ball bounces off a player.

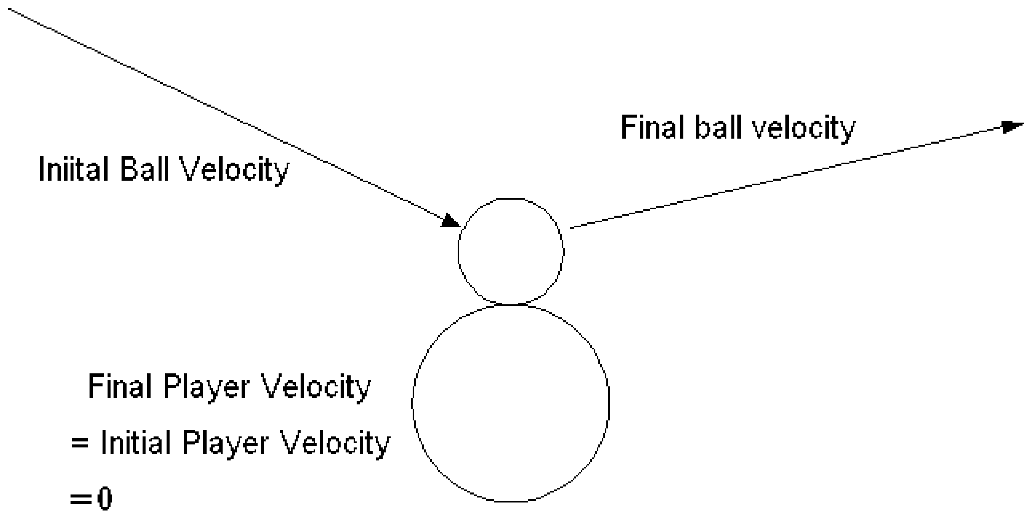

Figure 2: Ball hits motionless player.

Consider a reference frame in which the player is motionless and in which, at the time of the collision, the ball is directly above the player. See Figure 2.

The x component of the ball's velocity is unaffected by the bounce. The y velocity changes from down to up and its magnitude is multiplied by e where e is the co-efficient of restitution, which we take to be 0.5.

These considerations lead to the following algorithm for calculating the effect of ball, player collisions on the velocities of the ball and player.

Inputs:

- The initial velocity of the ball $\mathbf{v} = (v_x, v_y)$
- The position of the centre of the ball $\mathbf{a} = (a_x, a_y)$

- The initial velocity of the player $\mathbf{V} = (V_x, V_y)$
- The position of the centre of the player $\mathbf{A} = (A_x, A_y)$

Constant inputs

- The coefficient of restitution $e = 0.5$

Outputs

- The final velocity of the ball $\mathbf{v}'$
- The final velocity of the player $\mathbf{V}'$

1. Translate the ball velocity to the reference frame of the player
$$\mathbf{v1} := \mathbf{v} - \mathbf{V}$$
2. Calculate the angle of the vector from the player's centre to the ball's centre: θ
3. Rotate velocities so that the ball is right above the puck
$$\mathbf{v2} := \mathbf{v1} \text{ rotated counter-clockwise by } \pi/2 - \theta$$
4. Calculate the new velocity:
$$\mathbf{v3} := (v2_x, -e \times v2_y)$$
5. Rotate back
$$\mathbf{v4} := \mathbf{v3} \text{ rotated counter-clockwise by } \theta - \frac{\pi}{2}$$
6. Translate back to fixed reference frame.
$$\mathbf{v\prime} := \mathbf{v4} + \mathbf{V}$$

Note that in the unlikely case that the ball touches two players at once, you should consider this as two sequential collisions in either order.