The Ada Way Soccer Simulator

Ricardo Aguirre
Andrea Graziano
Marco Teoli
Alberto Zuccato

April 27, 2012

Abstract

"The Ada Way" is a European student challenge promoted by Ada-Europe. It has been chosen as project for the "Concurrent and Distributed Systems" teaching. It consists of a soccer simulator in which all the players and referees are computer-driven, while humans can interface to the system acting as coaches. Each player is an independent task and so it has its own thread of control. All players need to synchronize their movements and actions to play in a consistent manner.
In this document, presented according to the COMET methodology, we give an overview of the project design of the simulator, with special attention on the concurrency and distribution problems and on the solutions (syncronization on protected objects, access protocols etc.) we developed to deal with those issues.

Contents

I Problem specification and use cases **4**

1 Software requirements **5**
1.1 Introduction 5
1.1.1 Purpose 5
1.1.2 Scope 5
1.1.3 References 6
1.1.4 Overview 6
1.2 Overall description 6
1.2.1 Product perspective 6
1.2.2 Product functions 6
1.2.3 User characteristics 7
1.2.4 Constraints 7
1.2.5 Apportioning of requirements 7
1.3 Specific software requirements related to the simulation 8

2 Use case model **10**
2.1 Configure the simulation 11
2.2 Start playing 12
2.3 Abort match 12
2.4 Choose a team and configure its settings 13
2.5 Ask the referee for a substitution 13
2.6 Watch the players statistics 14
2.7 Watch the match 14
2.8 Check out the match statistics (teams only) 14
2.9 Catch the ball 15
2.10 Kick the ball 15
2.11 Tackle the ball carrier 15
2.12 Interrupt the game 16
2.13 Signal that the ball is out of bounds 17
2.14 Handle foul 17
2.15 Caution 18
2.16 Handle end period 18

II Analysis model **20**

3 Static model **21**
3.1 Static model of the problem domain . . . 21
3.2 Static model of the system context . . . 21
3.3 Static model of the entity classes . . . 23
3.4 Object structuring . . . 23

4 Dynamic model **26**
4.1 Configure the simulation . . . 26
4.2 Start playing . . . 27
4.3 Abort match . . . 27
4.4 Choose a team and configure its settings . . . 28
4.5 Ask the referee for a substitution . . . 28
4.6 Watch player statistics . . . 29
4.7 Watch the match . . . 29
4.8 Check out match statistics (spectators only) . . . 30
4.9 Catch the ball . . . 30
4.10 Kick the ball - ball out of bounds . . . 31
4.11 Tackle the ball carrier (with foul) . . . 31
4.12 Interrupt the game . . . 32
4.13 Handle foul (with caution) . . . 33
4.14 Assign substitutions . . . 33
4.15 Handle end period . . . 34

III Design model **35**

5 Design of the Soccer Simulator **36**
5.1 Consolidated artifacts . . . 36
5.2 Subsystem structure . . . 36

6 Distributed Component-Based software architecture **40**
6.1 Reasons to construct a distributed system instead of implementing a centralized system . . . 40
6.2 Design distributed component-based software architecture . . . 40
6.3 System configuration . . . 42
6.4 Startup execution order . . . 43
6.5 Summary . . . 45

7 Concurrent Design of the Core subsystem **46**
7.1 Design of Core subsystem . . . 46
7.2 Modelling the field, positions and motions in a discrete fashion . . . 49
7.3 Players as “intelligent” agents . . . 50
7.3.1 Player logic . . . 51
7.3.2 Determining player positions and movements . . . 54
7.3.3 Player actions . . . 55
7.3.4 Synchronization of player movements . . . 56
7.4 Players as concurrent tasks . . . 57
7.4.1 Synchronization of player movements . . . 57

7.4.2 Time synchronization of players with the ball 62
7.5 The status management and the referee staff 66
7.5.1 Status Manager 66
7.5.2 Assistant Referee 67
7.5.3 Referee 68
7.6 From players to referees: a matter of synchronization 70
7.6.1 Managing the game state 71
7.6.2 Detailed description of interactions and synchronizations . 72
7.7 Connecting the core with the world 78
7.7.1 The Frame Manager 78
7.7.2 The Event Manager 82
7.8 Initialization and finalization of core subsystem 85
7.8.1 Core initialization 85
7.8.2 Core finalization 87

8 Design of Billboard subsystem 89
8.1 Core Partition 89
8.2 BillboardOutput Partition 90
8.3 BillboardInput Partition 92
8.4 Java GUIs: 93
8.5 Distribution Summary (historical narrative) 96

9 Design of the distributed Graphical User Interfaces 99
9.1 High level design 99
9.1.1 Categorization of requirements 99
9.1.2 General considerations 100
9.1.3 GUI's initialization 101
9.1.4 GUI's normal operations 102
9.1.5 GUI's termination 103
9.2 Detailed design 103
9.2.1 Spectator GUI 104
9.2.2 Simulator GUI 105
9.2.3 Manager GUI 107

10 Conclusions 109

Part I

Problem specification and use cases

Chapter 1

Software requirements

The following description is based on the overall description of the software requirements recommended by the IEEE standard 830-1998.

1.1 Introduction

1.1.1 Purpose

This specification aims at outlining the requirements of our project, without pretending to present deep details about all of them.
More specifically, we give an overview of the functional requirements along with the non functional ones. For functional requirements we do not provide all the details because most of them will be clear afterwards, from the description of the use case model.

1.1.2 Scope

This project aims to realize a software architecture with characteristics of concurrency and distribution that simulates a soccer match. More specifically, the match should follow the rules of the classic 11-by-side soccer[1]. Players are computer-driven and should be able to determine by themselves the actions they want to carry out in order to play the game. The way they play also depends on their own physical characteristics. Any interruption (goals, fouls, balls falling outside the field etc.) should be handled by the referee and his assistant(s).
Humans can connect to the simulator using their user interfaces.
One user can act as simulator manager, who is allowed to choose the duration of each half, the duration of the half-time break and the maximum number of substitutions.
Two users can become team managers, i.e. they can change the default team configuration choosing the first-strings and substitutes of the game and the formation (that can also be changed during the match). During the match they can also ask the referee to substitute players up to the chosen maximum number

[1]Different game modes with less then 11 players per side are not supported. Other minor differences can be found comparing this specification with the original one published by Ada-Europe.

of substitutions.
Many other humans can connect to the game as spectators. People should be allowed to connect to the game both locally or from a remote computer. The simulator should work with or without any external intervention. This means that the connection of the simulator manager and team managers should be allowed but is not required.

1.1.3 References

This specification is based on the official requirements provided by Ada-Europe[1]. The simulator follows the "Laws of the Game" published by FIFA in 2010 [5].

1.1.4 Overview

In the next section we provide an overall description of the system requirements. A more detailed specification - only for the requirements related to the simulation of the match - is provided in the following section. The presentation of some requirements is deferred to the second chapter of this report, which presents the use case model according to the COMET methodology.

1.2 Overall description

1.2.1 Product perspective

The Soccer Simulator shall:

- have a centralized software core, implementing all of the logic of the simulation;
- provide an interface for the display of the match; players can be displayed as in a view of a Subbuteo game seen from the top. Many instances of such interface shall be available on request;
- provide an interface for the users to let them act as team managers; up to two instances of such interface can be available, which shall display the current parameters for each player;
- one read-only graphical panel (window) for the display of the match statistics (for teams and players).

The system shall run out of the box on GNU/Linux systems. Other operating systems can be supported.

1.2.2 Product functions

F1 Users must be able to play a single game.

F2 The simulator shall implement the canonical 11-a-side format. More details about the level of support given to the game format will be provided in the next section. In addition:

(a) the simulator user shall be allowed to configure the maximum number of substitutions (three or six for each team);

(b) the simulator user shall be allowed to configure the durations of the game and of the half-time break.

F3 The members of the squads will feature individually configurable characteristics and play according to them.

F4 The simulator shall be able to configure the initial players' line up and the initial tactic without any human intervention.

F5 The match shall have one independent (software) referee and a subordinate (software) assistant; they control the game and ensure that the applicable rules are followed.

F6 The behaviour and the performance of the referee and assistants need not exhibit the physical limitations of actual humans.

F7 (optional) support for playing a series of matches, with fixtures and associated rules.

1.2.3 User characteristics

Four classes of users can be identified:

- a Simulator user, who configure the match settings (number of substitutions and durations); he/she is allowed to bring forward the start of a half and to abort the game;
- (team) Manager users, who act as coaches for the playing teams;
- Spectator users, who connect to the game to watch the match;
- the system administrator, who configure the system.

Simulator, managers and spectators have an average knowledge of computer use and soccer laws. The system administrator is someone who has the ability to configure a distributed system following the instructions given with the software.

1.2.4 Constraints

1. The software core shall be programmed in Ada.
2. Some parts of the system shall be possibly programmed in other programming languages.

1.2.5 Apportioning of requirements

Requirements shall be apportioned incrementally. Aspects that are not made explicit (such as the implementation of some specific law of the game) can be delayed to future versions of the system.
In determining what should be done at each time, the following criteria should be adopted:

- the first version shall provide an effective simulation of the match, e.g. a simulation should run until the end of the game without evident misbehaviour;
- the development of the laws of the game can be delayed and/or left out whenever simulation actors (players, referees) are not supposed to abuse of non-implemented laws. For example, off-sides and required distances within players can be left out if players are not programmed/able to take advantage of their absence;
- aspects that are not covered by/relevant to our model of the game can be ignored. For example there is no need to model the shape and weight of the ball, nor the equipment of players and referees, because players will be shown as in a Subbuteo game seen from the top and the referees will not even be shown.

1.3 Specific software requirements related to the simulation

The soccer simulation should support most of the possible events that can happen during a real match.

S1 Players shall support the following characteristics:

(a) speed;
(b) passing power and accuracy;
(c) shooting power and accuracy;
(d) height, jump skill, heading power and accuracy;
(e) tackle/dribble skill, dribbling and catching accuracy;
(f) aggression;
(g) stamina (the strength that the player is able to oppose to fatigue).

S2 Players have a dynamic fitness status that decrease during play because of fatigue according to their stamina.

S3 Players' behaviour shall support the following abilities:

(a) kick/throw-in the ball;
(b) stop/catch the ball;
(c) tackle/dribble an opponent;
(d) determine a movement to put himself in a convenient position;
(e) move toward a chosen direction;
(f) make fouls;
(g) determine the correct action according to notifications received during the game (i.e. decide to move outside the field if sent off or injured, determine whether he should restart the play after an interruption and eventually decide to go toward the ball).

S4 The effectiveness of players' actions shall depends on their physical characteristics.

S5 The following Laws of the Game shall be supported by the simulator:

1 "The Field of Play" (with dimensions rounded to integer values), which dimensions shall be of 105 m · 68 m;

3 "The Number of Players". Each match shall have 17 participants for each team (11 first-strings, 6 substitutes); substitutes are not required to play;

5 "The Referee", who guarantees the application of the laws implemented by the system;

6 "The Assistant Referee"; only one assistant referee is actually required to check all the lines, because; referees do not need to exhibit limitations of humans; assistant referee's "gestures" can be developed as messages that tell the referee what kind of interruption has to be handled;

7 "The Duration of the Match", that shall be possibly changed on request by the Simulator User before the beginning of the game; it does not need to be set according to the laws of the game;

11 "Offside" shall be at least partially supported, i.e. players should not take offside positions except for unwanted exceptional situations; otherwise the referees should check offsides;

12 "Fouls and Misconduct". Only fouls (and their handling) shall be implemented; complex situations can be avoided;

13 "Free Kicks". For simplicity, only direct free kicks shall be supported. Adoption of indirect free kicks is optional;

14 "The Penalty Kick";

15 "The Throw-in";

16 "The Goal kick";

17 "The Corner kick".

Chapter 2

Use case model

The supported use cases, illustrated in figures 2.1 and 2.2, are described in the next paragraphs.
Primary use cases are instantiated by humans. Other use cases are started by players or referees during the match. We can see all of them as a "consequence" of the beginning of the match.

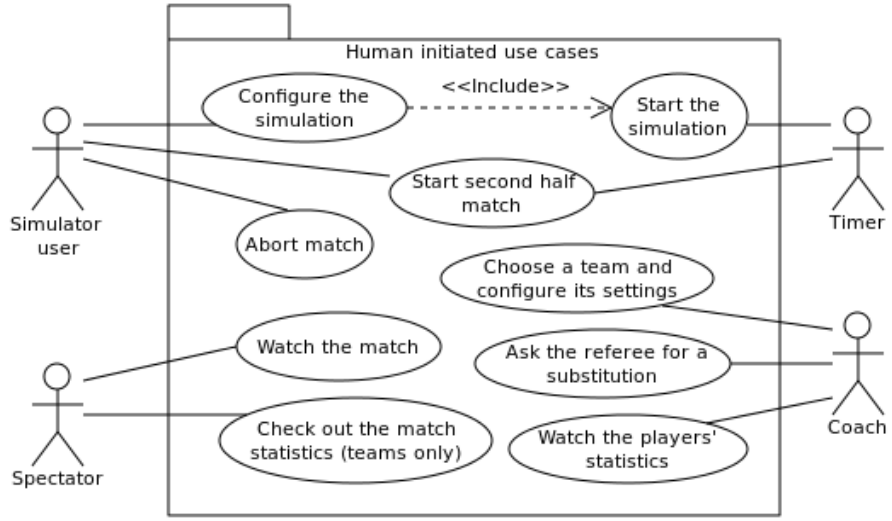

Figure 2.1: Primary use cases

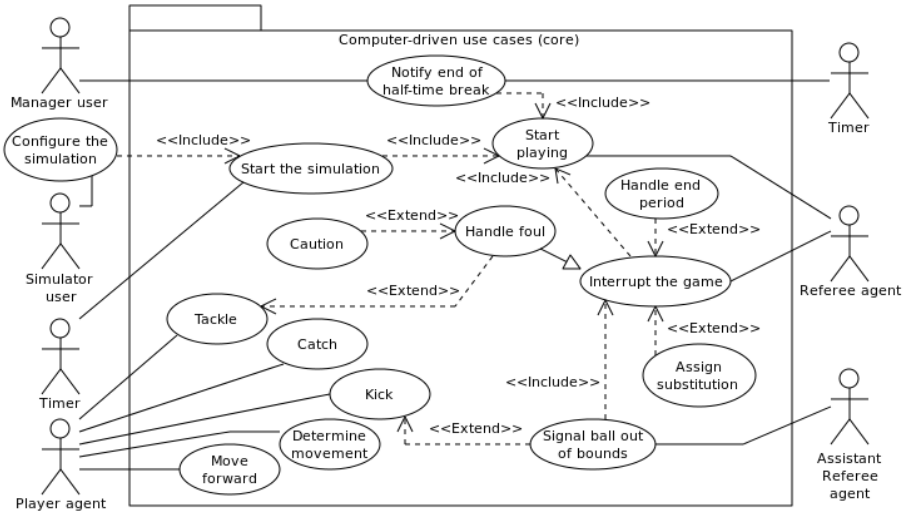

Figure 2.2: Computer-driven use cases (core)

2.1 Configure the simulation

The actor of this use case is the "Simulator user"; he/she is supposed to configure a new match and to start the game afterwards.

Summary The simulator manager user configures the parameters of the simulation and lets it start.

Actor Simulator user.

Precondition The System has been started and the time-out for the match configuration has not been expired.

Description :

1. User connects to the System.
2. The System accept the connection of the simulator user because it is the first one (and the only allowed).
3. User sets up the match configuration (duration of halves and half-time break, maximum number of substitutions).
4. User requests the start of the simulation.
5. The computer-driven referee tells the players to position for a kick-off.
6. Include "Start playing" use case.

Alternatives :

- If no user connects, the system starts the simulation after a predefined time-out has expired.
- If another Simulator user is already connected to the System then this new connection is refused and the new user will not be allowed to continue.

Postcondition The request has been accepted and the System proceeds starting the simulation.

2.2 Start playing

Summary The simulator user request to anticipate the beginning of the match.

Actor Simulator user (secondary actor: referee).

Precondition The referee has signalled how to start playing.

Description

1. The teams sets play (in the simulation, this will just mean that the player who has to restart has to be determined somehow).
2. The referee waits until players take position.
3. The player who has to start playing goes next to the ball. The other players position themselves according to the restart mode indicated by the referee.
4. When all the players are in their right position, the referee whistles (this simplifies the roles, because it means every players always have to wait for all the other players, opponents included, even when they could take advantage).
5. The starting player decides his target man and prepares to pass him the ball.
6. Include "Kick the ball".

Postcondition The match has started (over).

2.3 Abort match

Summary Simulator User decides to abort the match.

Actor Simulator user.

Precondition The referee whistled the start of the match.

Description

1. The Simulator User requests the abortion of the match.
2. The system accept the request and stops the simulation.

Postcondition The game is over.

2.4 Choose a team and configure its settings

Summary A team manager selects his or her team for the match.

Actor Manager User.

Precondition The countdown before the match has started, but not the match itself. The manager opened his/her own interface.

Description

1. Manager user connects to the system.
2. The user chooses whether he wants to play as local or visitor.
3. The user chooses a team.
4. The user configures the initial line up and tactic.
5. The user confirms the team settings.

Alternatives

- If the manager user does not complete his team configuration in time, the team settings will not be accepted and he will play with the defaults (i.e. he will be assigned a default team with a predefined line up).

Postcondition After the configuration time-out, both team configurations are assigned (according to manager choices, if received, or otherwise with defaults values).

2.5 Ask the referee for a substitution

Summary A team manager substitutes one of his players.

Actor Manager user (coach).

Precondition The match has already started; the team manager is connected to the system.

Description

1. The Manager User requests a substitution.
2. The referee receives, validate and registers the substitution.

Alternatives

- If the substitution cannot be accepted (according to the laws of the game), the substitution will not be registered (or otherwise it will be refused when handled).

Postcondition The requested substitution, if acceptable, has been registered by the referee.

2.6 Watch the players statistics

Summary Manager user checks the statistics and status of his players.

Actor Manager user.

Precondition Manager user is connected to the system.

Description

- The soccer simulator periodically sends statistics of each player to the corresponding manager user.
- The Manager user checks the statistics on his own interface.

2.7 Watch the match

Summary Spectators connect and watch an ongoing match.

Actor Spectator User.

Precondition The Soccer Simulator is up and running.

Description

1. Simulator user connects to the system.
2. The Soccer Simulator sends snapshots of the game to the Simulator user display at the default frequency.

Alternatives

- If the Spectator user changes the update frequency of his/her display (television), the new value is sent to the Soccer Simulator that starts providing the snapshots at the desired frequency.

2.8 Check out the match statistics (teams only)

Summary Spectators check match statistics.

Actor Spectator user.

Precondition Spectator user is already connected to the system.

Description

1. The software simulator periodically sends updated statistics to spectators' interfaces.
2. Spectator user presses the "Show statistics" button.
3. The user interface shows the statistics received and keeps them updated (whenever new statistics arrive).

2.9 Catch the ball

Summary A player catches a stationary ball or attempts to stop a moving one.

Actor Player.

Precondition The player does not have possession, the ball is free (moving o stationary) and close enough to be stopped.

Description

1. The player tries to stop a moving ball or take possession of a stationary one.
2. The catch attempt succeeded, the ball is on the player's feet.

Alternatives

2.1 The player's attempt to catch a moving ball failed, the ball keeps running.

2.10 Kick the ball

Summary A player passes the ball to a teammate or shoots it into the net.

Actor Ball carrier.

Precondition The ball carrier realizes that he could pass or shoot the ball (he saw a team-mate in a convenient position, an opponent ready to attempt a tackle or some chance to score a goal).

Description

1. Kick the ball toward a team-mate or at the opponent's goal, as chosen.
2. The ball moves toward the direction impressed by the player.
3. The ball is stopped by another player or stops inside the field by itself.

Alternatives

3.1. The ball falls out of bounds, eventually into one of the nets.

3.2. Include "Notify that the ball is out of bounds".

Postcondition The kick has thrown the ball away. If the ball has fallen out of bounds, the event is detected and handled by the referees.

2.11 Tackle the ball carrier

Summary A player attempt to take the ball away from the ball carrier.

Actor Player (secondary actor: ball carrier).

Precondition The defending player and the ball carrier are closed enough to each other.

Description

1. The player attempts to tackle the ball carrier.
2. The referee verifies that the tackle was fair.
3. Player obtains possession of the ball.

Alternatives

2.1. The referee realizes that the attacking player committed a foul.

2.2. Include "Interrupt the game".

3.1. Player does not obtain possession of the ball. The opponent maintains possession.

Postcondition The tackle attempt succeeded or failed, any penal foul and caution has been assigned and the play started over.

2.12 Interrupt the game

Summary The referee stops the game because the ball has fallen out of bounds or a foul has happened. Then he manages the restart of the game.

Actor Referee (secondary actors: players).

Precondition Some condition exist that requires the referee to stop the game.

Description

1. The referee whistles to stop the game.
2. The referee verifies that the interruption was due to a ball falling out of the bounds.
3. The referee verifies that the period of play is not yet finished (or otherwise that no throw-in/goal kick condition applies).
4. The referee verifies that there are no pending substitution requests.
5. The referee signals (by hands) how the play will be restarted.
6. Include "Start playing".

Alternatives

2.1. The game was stopped for a foul, include "handle foul". Then continue with step 4 of the main sequence.

3.1. The game was supposed to restart with a throw-in or a goal kick but the time for the current period is already finished, include "end period".

3.2. Continue from step 6 of the main sequence (this does not happen if the second period is over).

4.1. The referee validates pending substitutions requests and notifies the managers whether their requests were accepted or not.

4.2. Substituted players leave the field, substitutes enter.

Postcondition The interruption has been managed and the game started over.

2.13 Signal that the ball is out of bounds

Summary The assistant referee sees the ball falling out of bounds (possibly inside a goal) and notifies the referee.

Actor Assistant Referee (secondary actor: Referee).

Precondition The ball is out of bounds.

Description

1. The assistant referee sees the ball falling out of the field area.
2. The assistant referee determines which condition applies:
   - the ball is inside one of the goals and was played by the attacking team (goal);
   - the ball is inside one of the goals and was last touched by the defending team (own goal);
   - the ball has fallen behind one of the goal lines (but outside of the goal) and was played by the attacking team (goal kick);
   - the ball has fallen behind one of the goal lines (but outside of the goal) and was played by the defending team (corner kick);
   - the ball has fallen behind one of the sidelines (throw-in).
3. The assistant referee notifies the referee of the occurring condition.
4. include "Interrupt the game".

Postcondition The detected event caused an interruption, which has been correctly handled by the referee.

2.14 Handle foul

Summary The referee awards a free kick or a penalty kick because of a penal foul.

Actor Referee.

Precondition The offending player committed a penalty foul.

Description

1. The referee sees that the foul committed is not a sanctionable offence.
2. The referee whistles for a free kick (or a penalty kick if the foul was done by a defender inside the penalty area).
3. Include "Start playing".

Alternatives

1.1. The referee sees that the foul should be sanctioned.

1.2. Include "Caution" (then continue from 2.).

Postcondition The free/penalty kick has been awarded, any due sanction assigned and the play has been started over.

2.15 Caution

Summary The referee sanctions the offending player.

Actor Referee.

Precondition The referee has detected a penal foul committed by a player and has consequently interrupted the game.

Description

1. The referee verifies that this will be the first sanction for the offending player and that the severity of the foul is not such as to require a send-off sanction.
2. The referee cautions (yellow card) the offending player.

Alternatives

1.1. The referee sees that this one is the second caution for the offending player or determine that the severity is such that the player deserves to be sent-off.

1.2. The referee assigns a send-off sanction to the offending player.

Postcondition The offending player has been correctly sanctioned by the referee.

2.16 Handle end period

Summary The referee stops the game because the current period of play is finished.

Actor Referee.

Precondition The current period of play expired and the ball went out of bounds across a sideline (or the goal line, outside the goal, kicked by the team on offence).

Description:

1. The referee whistles for half-time break because the last period was the first one.
2. The players leave the field.
3. Teams switch side.
4. The referee waits until he is notified about the end of the half-time break (by the Simulator Manager user or by the break time-out); in the mean-time, he acceps any valid substitution request.
5. The referee call the players inside the field for a kick-off.
6. Include "Start playing" use case.

Alternatives

1.1. The last period was the second one, the referee whistles the end of the match.

1.2. Players leave the field. The game is over (exit this use case and with any including one).

Postcondition The second period has started after the half-time break; in case the ended period was the second, everybody has left the game.

Part II

Analysis model

Chapter 3

Static model

3.1 Static model of the problem domain

The possibility of playing a league or a set of different matches is an option and will not be considered in the current version of this project/document.
There are a set of physical entities of the system that can be easily found.
The game is played in a field of fixed dimensions, made by a surface divided into many different areas with lines, spots and goals. During the game there will always be a ball inside (or near) the field.
A match is a challenge between two teams. The match itself is not a physical entity of the system, but we pictured it as an element of our problem domain because it helps in showing the existing relationship between all the other entities of the system.
Teams have a manager (or coach) and a fixed number of players. The manager is supposed to choose the first strings (11 players) and the substitutes (7 players). If a team has no human managers connected, the system loads a predefined initial line-up of the squad. Players are computer-driven, so no human-computer interaction is needed to have them playing.
The match is then directed by a computer-driven referee helped by one (again, computer-driven) assistant referee. Many spectators can watch the match, while another human can optionally control the simulation (start/abort).

3.2 Static model of the system context

The soccer simulator system interacts with many different external interfaces to permit interactions with humans. One simulator user can connect to the system to set up the configuration of the match and to start the simulation. If there is no one connected as a simulator user, then the system starts the simulation as soon as its time-out expires.
Up to two humans can connect to the system to set-up the team configuration and to ask the referee for substitutions during the play. Managers can change the tactic during the game and check statistics and physical characteristics for each of their players. Teams with no human manager will be configured with a default initial line-up.
The system also supports an indefinite number of spectators, who also have

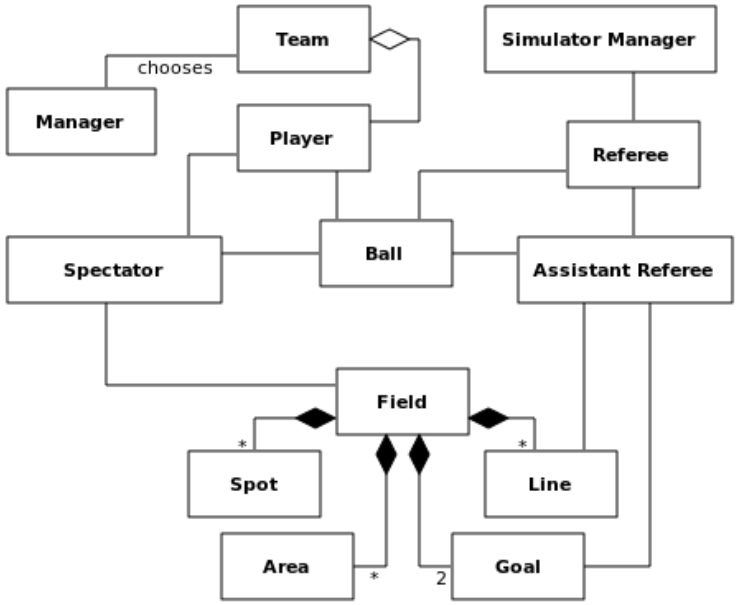

Figure 3.1: Conceptual static model of the problem domain (focus on physical entities)

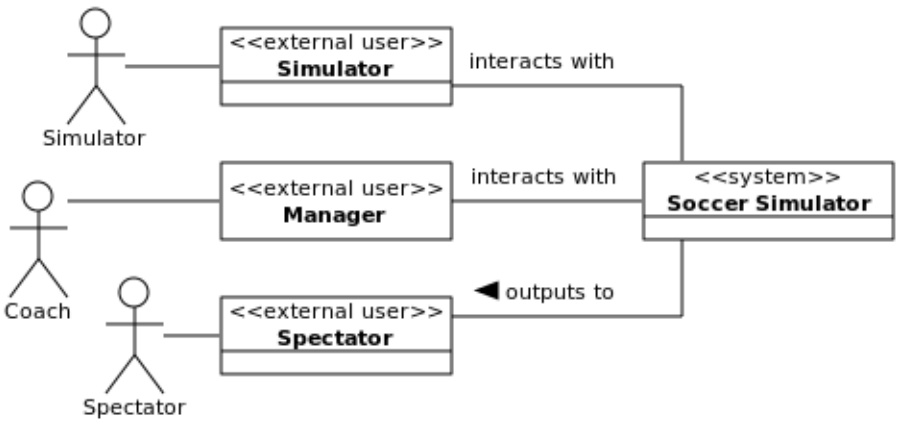

Figure 3.2: Context class diagram

their own user interface.
The system outputs to the spectator interface:

- the events of the simulation to let users watch the game;
- the overall statistics of the match for both teams.

3.3 Static model of the entity classes

Starting from the problem domain we identified a set of physical entities and gave an overview of the context in which the system is expected to work. The system under analysis is plenty of different entity classes.
Managers connected to the system can send substitution requests to the referee. Sanctions (yellow and red cards) can be instead assigned by the referee to players. Each player has his own identifier and characteristics. Some of them are static (name, shirt number, skills, stamina, ...) others are dynamic (fitness). Dynamic characteristics affect the quality of the play of each player. Each player also has his own status that tells for example if he has been injured, substituted or sent off. They also have a position inside the field and they follow a (linear) motion while moving to a target position. They belong to a team that also have some configuration (the formation in use).
The ball moves in a way that is somehow similar to that of players, but it does not have an AI: its movement depends on actions made by players. The ball has a status, a three-dimensional position that captures where it is in a particular moment and it follows a parabolic motion after being kicked. The players and referees, according to their own logic, take actions that generate events that need to be shown to spectators (manager users also act as spectators in that they also need to watch the game).
A number of statistics about the player conditions and about the actions they made are collected along with team statistics that usually sum them all. These statistics have to be shown to managers (player specific statistics) and spectators (team statistics).

3.4 Object structuring

Figure 3.4 shows the Soccer Simulator system. The soccer simulator can have many users connected; an indefinite number of spectators should be supported. This means that they would watch the match at the same time and so many instances of user interfaces can be up and running at the same time. All these interfaces, depicted in Figure 3.5 receive data from the soccer simulator. The Simulator User needs an interface to interact with the system (start/abort the simulation, set-up its configuration). That interface is obviously part of the soccer Simulator System. The same holds for spectators and managers (coaches). Managers are generally also supposed to act as spectators because they need to watch the game to decide how they want to manage their teams.
All the entity classes depicted in Figure 3.3 clearly belongs to the Soccer Simulator system or to its subsystems, which are identified in early design description.

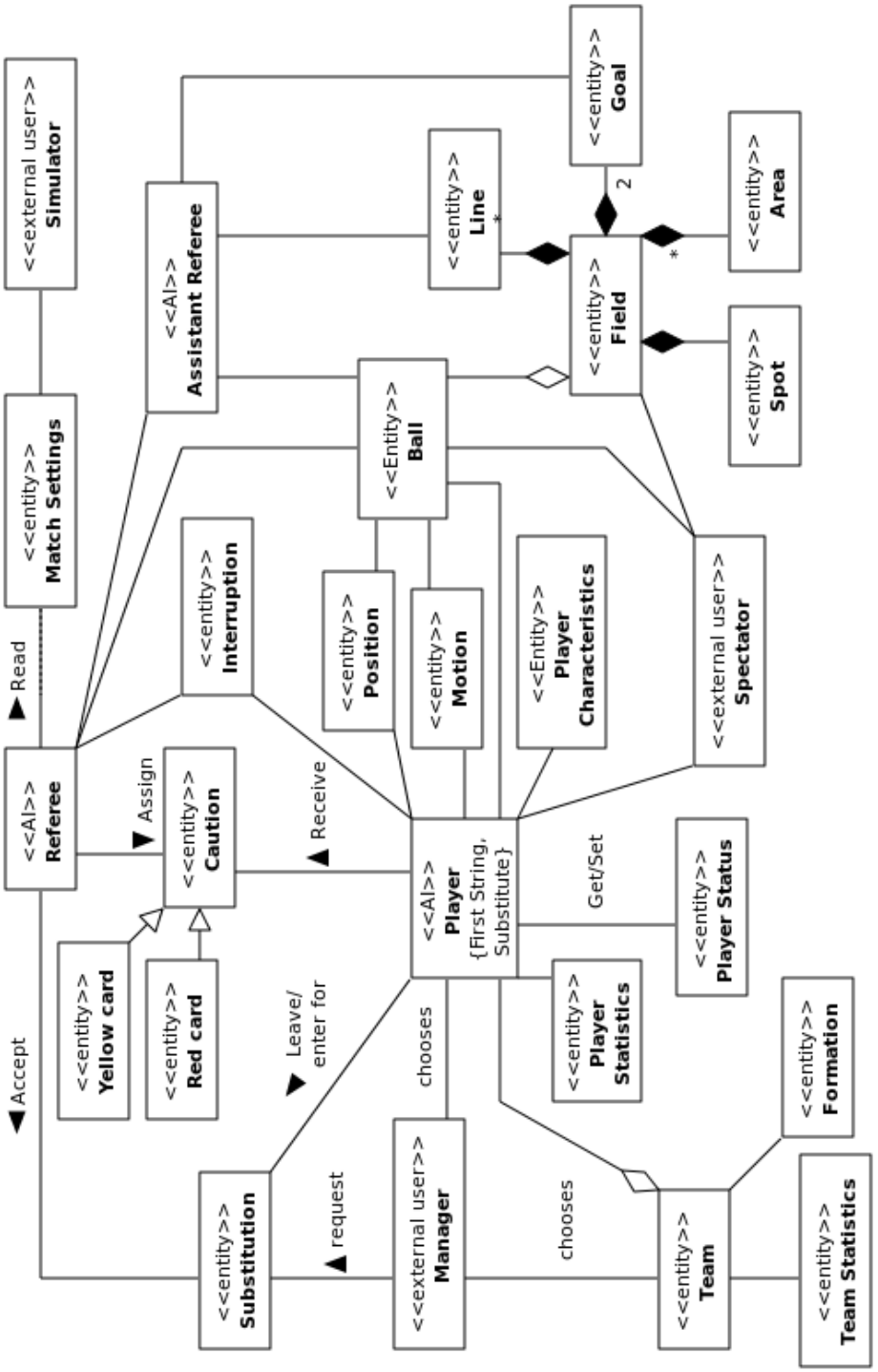

Figure 3.3: Entity classes

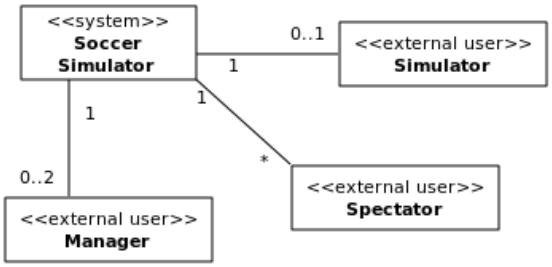

Figure 3.4: System structure

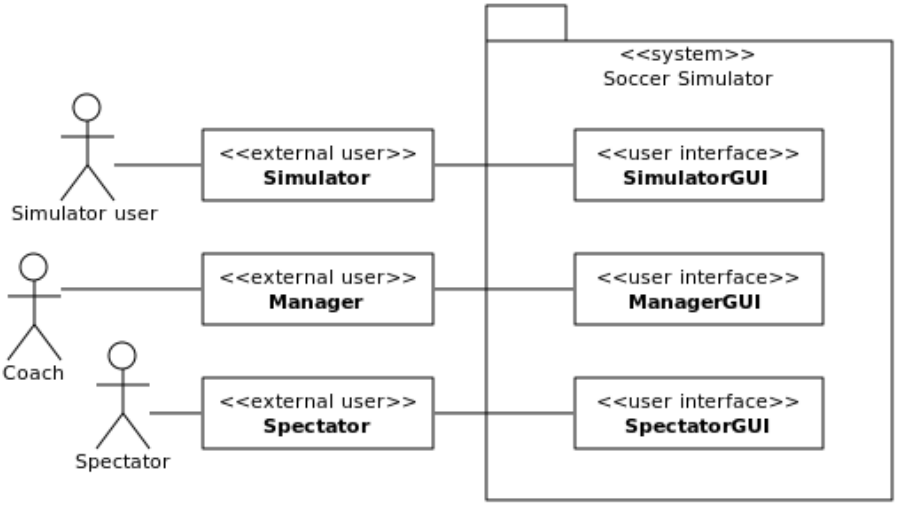

Figure 3.5: Soccer Simulator system: external classes and interface classes

Chapter 4

Dynamic model

This chapter shows the interactions that realise the use cases previously described in the use case model.

4.1 Configure the simulation

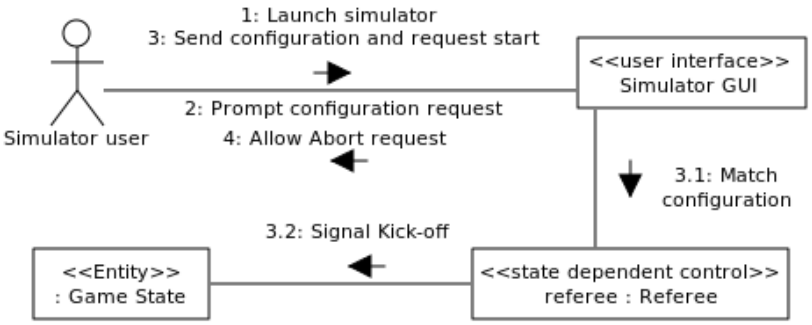

Figure 4.1: Communication diagram for the "configure the simulation" use case

1 User launches the Simulator user interface. The Simulator GUI is not the simulator core, which was already running.

2 Simulator GUI shows a prompt in which the user is asked to input the match configuration (duration of halves and interval, allowed number of substitutions).

3 Simulator user types in the match configuration and presses the start button.

3.1 Simulator GUI sends the match configuration to the Core's referee and ignores any further "Start Match" request.

3.2 Referee configures the simulation with the values received and signals players to position for a kick-off.

4.2 Start playing

Before the start, teams should set play. In our simulator, setting play consists only on determining which player should start playing. Anticipating a design choice, the request to the proper team to set play could be done by the referee. Afterwards, the referee signals the restart mode and wait until players are ready. In the meantime, players reach their position and wait the referee whistle. As soon as this happens, the startikng player passes the ball to a teammate.

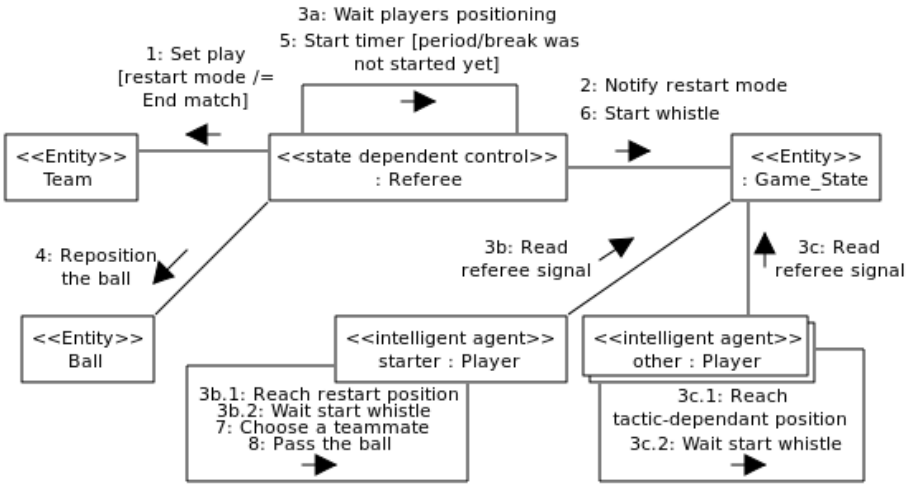

Figure 4.2: Communication diagram for the "Start playing" use case

4.3 Abort match

When the Simulator user requests to abort the game, the notification arrives to the referee who change the game state to "Aborted". This is obviously an extra signal compared with those that exist in soccer.
Players do not leave the field, they just leave the game by signalling that they are done. The referee waits until all the players leave and then signals to his assistant that the match is over. This sequence is not necessary (they could all finish executing immediately), but introduce in principle something that will be necessary on finalization design.

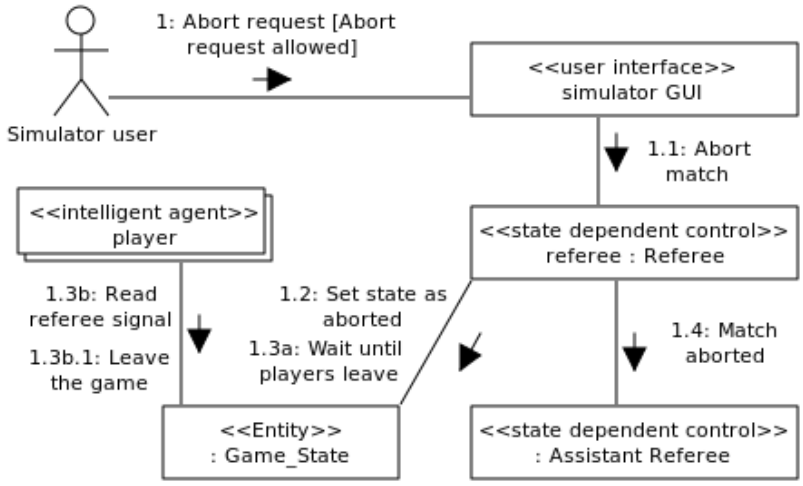

Figure 4.3: Communication diagram for the "Abort match" use case

4.4 Choose a team and configure its settings

The figure shows how a Manager user connects to the system, choose his or her team and configure the initial line-up and formation. The manager user do it by adapting the default one shown by his/her user interface.

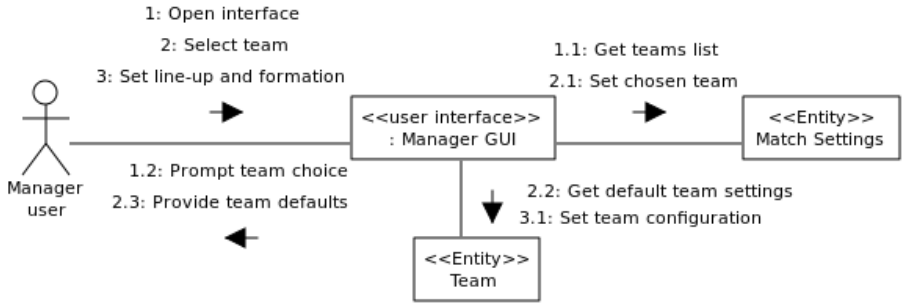

Figure 4.4: Communication diagram for the "Choose a team and configure its settings" use case

4.5 Ask the referee for a substitution

The Manager user can request a substitution that will be registered by the referee and then managed during the first interruption only if admissible. The diagram shows only the registration phase. The admissibility could also be checked at that time to avoid useless registrations, but must be checked again just before handling it, because in the meantime the player now being substituted could have been sent off.

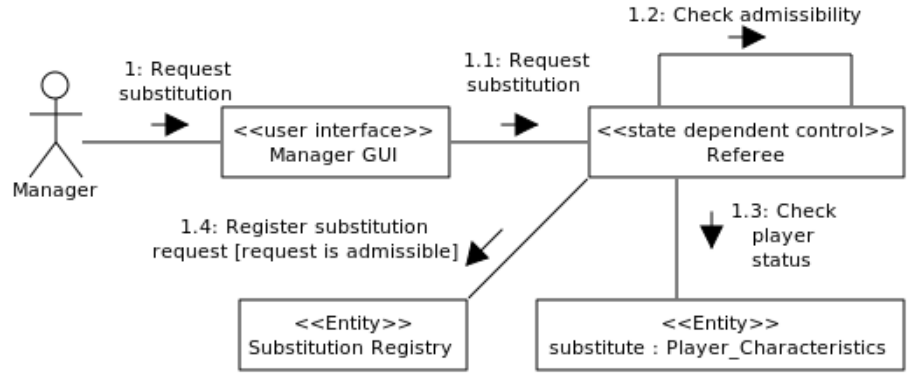

Figure 4.5: Communication diagram for the "Ask the referee for a substitution" use case

4.6 Watch player statistics

The manager can watch the statistics and status of all his/her players; no direct interaction with the user is needed: an entity of the system, called "Statistics Manager", periodically reads players' statistics and sends them to the user interface; the data received is then periodically refreshed on the display so that it can be read by the user.

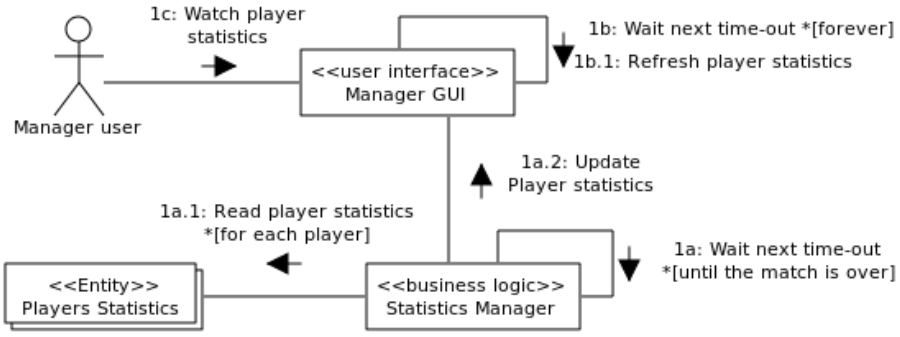

Figure 4.6: Communication diagram for the "Watch player statistics" use case

4.7 Watch the match

This interaction is analoguous to the one that describes the Manager user watching statistics, but in this case the Frame Manager reads the positions of the ball and players to create a snapshot of the game and keeps doing this periodically until the match is over and all the players have left the field. A manager user, in order to watch the game, must also act as spectator and connect through the Spectator GUI.

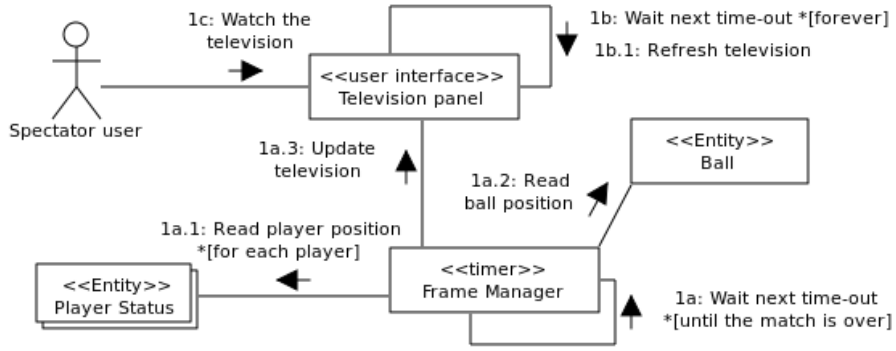

Figure 4.7: Communication diagram for the "Watch the match" use case

4.8 Check out match statistics (spectators only)

Spectators can see, in a table, aggregated statistics that sum up the players statistics for each team. This work as in the previous cases with the only difference that spectators are supposed to open a different panel to watch the aggregated statistics.

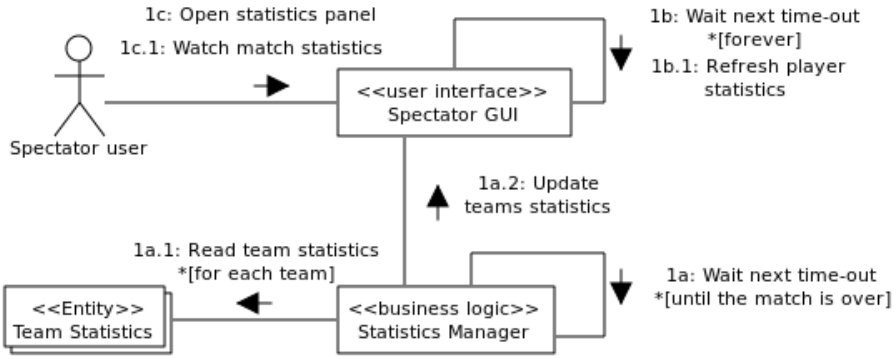

Figure 4.8: Communication diagram for the "Check out match statistics" use case

4.9 Catch the ball

Players actions are made up of two phases: calculating the action effects and applying the consequent changes. A player who wants to stop/catch the ball executes a business logic that calculate the motion that will be impressed to the ball (the motion is different from player's expectations accordingly to his own skills).
Then the player delegates the Status Manager to execute the action on his behalf. Introducing the Status Manager we anticipate a design choice needed in order to do separation of concerns between players and facts that happen to the status in consequence of players' actions. The main idea is that we would

like to avoid excessive coupling between players and the status of the game.

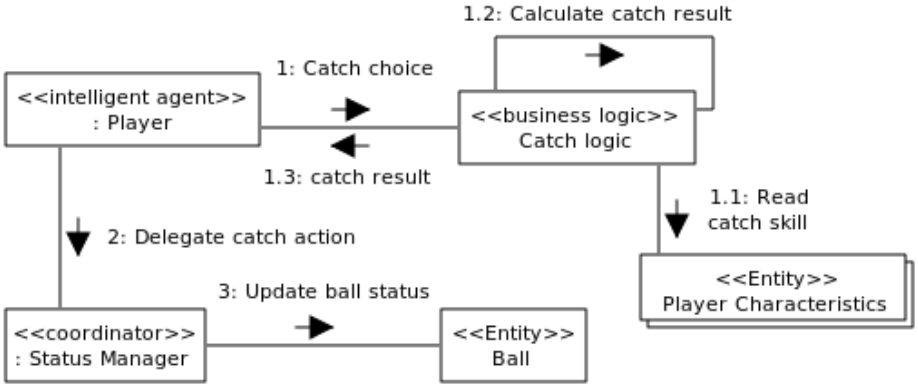

Figure 4.9: Communication diagram for the "Catch the ball" use case

4.10 Kick the ball - ball out of bounds

All the actions that can be done by players follow a similar sequence of steps. When a player kicks the ball, it might fall out of bounds. In that case the assistant referee (the linesman) detects what happened and notifies the referee of the occurring condition. We described an assistant referee who watch the lines, as it happens in reality, but from a design perspective this could lead to polling. Consequently, we might prefer that the event is notified by others to the assistant referee than viceversa.

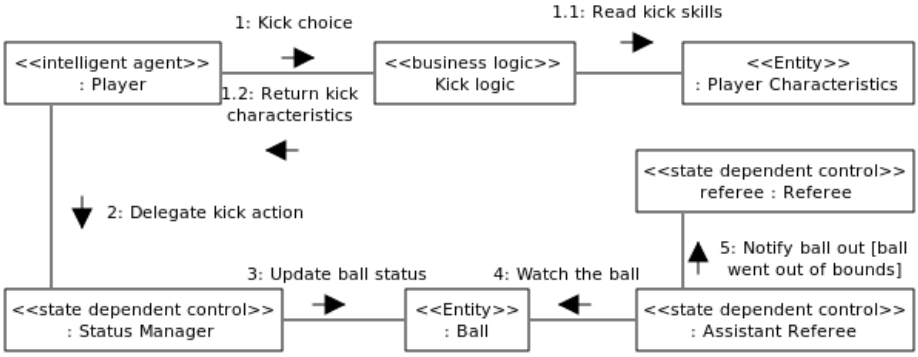

Figure 4.10: Communication diagram for "Kick the ball" and "Ball out of bounds" use cases

Condition for steps 4-5: the ball went out of bounds from one of the sidelines (throw-in condition) or it has been sent out from one of goal line by the attacking team (goal kick condition).

4.11 Tackle the ball carrier (with foul)

Tackles follow the scheme we described for kicks. An eventual foul is in this case registered by the status manager and detected by the referee.

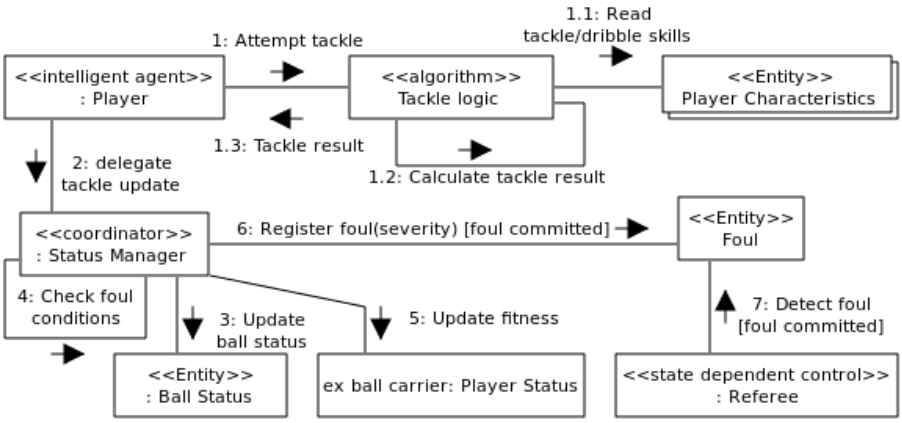

Figure 4.11: Communication diagram for the "Tackle the ball carrier (with foul)" use case

4.12 Interrupt the game

This sequence is executed after the referee is notified that the ball has fallen out of bounds.

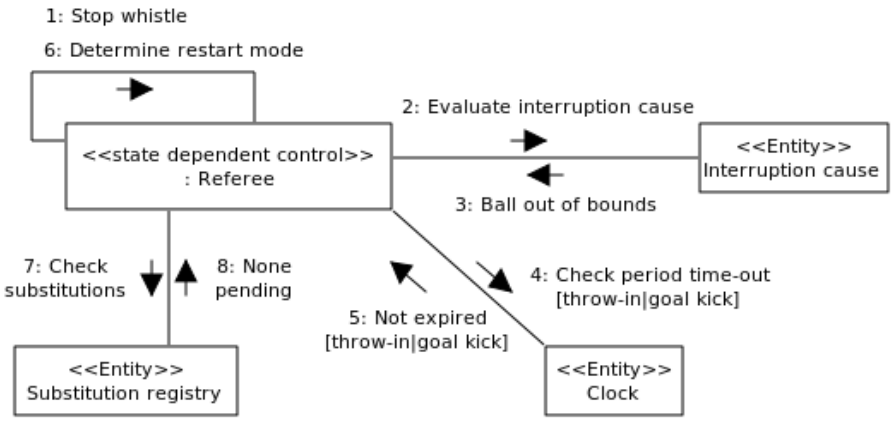

Figure 4.12: Communication diagram for the "Interrupt the game" use case

A similar sequence can be executed for other types of interruptions. For example, in case of foul the cause of interruption is different but those steps are executed as well. The sequence is expanded, if needed, to support some extra condition, such as penalty fouls and pending substitutions.
Note that:

- If the interruption is due to a foul, the "handle foul" sequence precedes the calculation of the restart mode (step 6);
- If the time available for the current period has expired, executes the "handle end-period" sequence and interrupt this sequence;
- If some substitution is pending, the "assign substitutions" sequence anticipates the restart of the game ("start playing").

4.13 Handle foul (with caution)

This interaction is executed during interruption handling in case the interruption was due to a penal foul. Note that before whistling to start over, the referee has to wait until all players take position.

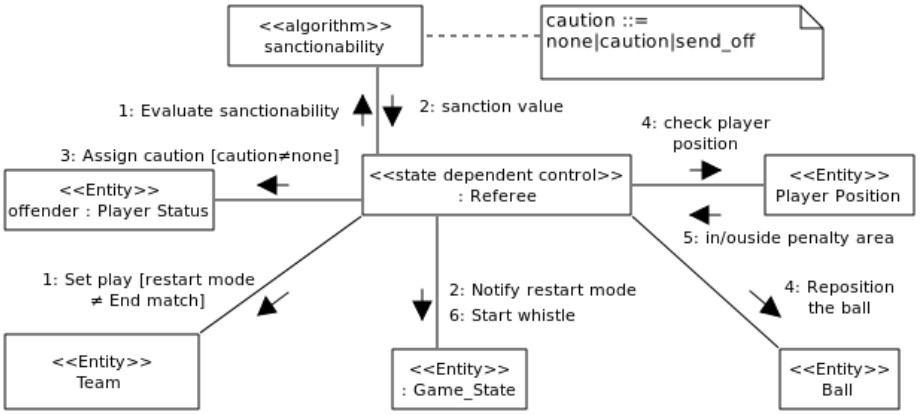

Figure 4.13: Communication diagram for the "Handle foul" and "Caution" use cases

4.14 Assign substitutions

This interaction occurs during interruption handling in case a team manager previously requested a substitution. As we already wrote, whether or not the admissibility of the substitution have already been checked before, it must be checked in this phase as well.

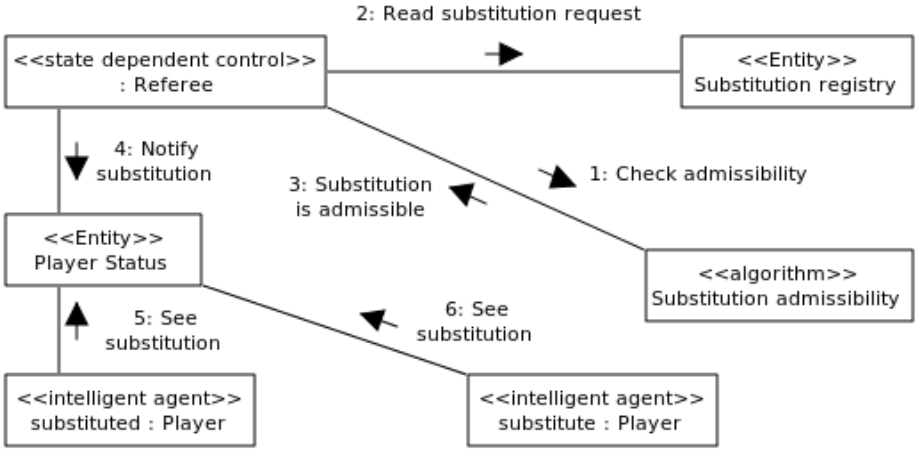

Figure 4.14: Communication diagram for the "Assign substitution" use case

4.15 Handle end period

When the referee stops the game because the ball has fallen out of bounds, in case of throw-in, goal kick or kick-off, he also checks whether the time for the current period of play is over.
If the first half is finished, he signals the event and wait until players leave the field, then he waits for the duration of the half-time break and signal kick-off, so that players re-enter the field. The message "Field left" means that - because the referee sees from the game state when players leave the field - players are supposed to change the state in order to notify that they left.
If the second half is finished, the sequence is analoguous but the steps 3.* are skipped.

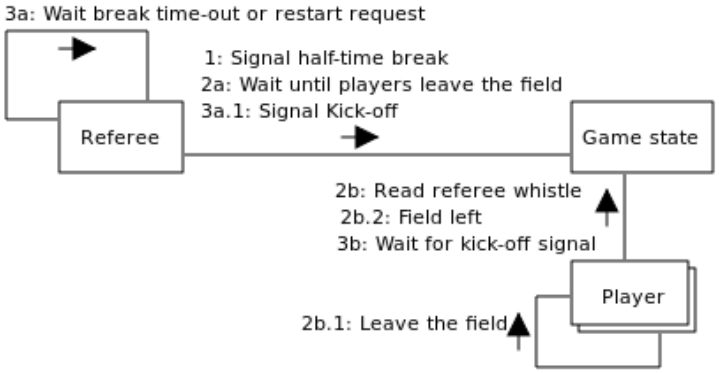

Figure 4.15: Communication diagram for half-time break case of the "Handle end period" use case

Part III

Design model

Chapter 5

Design of the Soccer Simulator

5.1 Consolidated artifacts

So far we described the interactions that happen when use cases are executed. Figure 5.1 shows the consolidated artifacts of the whole system.
We simplified some aspects to avoid overcrowding the picture. "Action logic" object substitutes "Catch logic", "Throw logic" and "Tackle logic" objects. Some parts of the system will be refined in the next chapters in order to present a convenient task architecture.

5.2 Subsystem structure

Excluding user interfaces, all the entities shown in the previous section belong to the same subsystem, that we called "Core". This design choice is partially due to the coupling of those entities, and partially depends on our choice of running the whole simulation on one node.
Hiding the entities that do not interact with the external interfaces, our system could be deployed as shown in Figure 5.2.

Many instances of user interfaces can be up and running at the same time. More specifically, there might be one Simulator User, two Manager Users and an indefinite number of Spectator Users, all of them with their own user interface. The Core subsystem has to send updates to all these interfaces; it also receives requests (from Simulator GUI and Manager GUI) and connections.
Some extra component is needed to handle all these communications and we decided to deploy them in an object broker subsystem that we call "Billboard". The Billboard subsystem is expected to manage communications between Core and GUIs in a publish/subscribe fashion. All the users that want to connect to the simulation must register to Billboard through the connection and validation of their user interface.
Moreover, different user interfaces (or different instances of the same one) can register to receive different information from the Billboard (or the same infor-

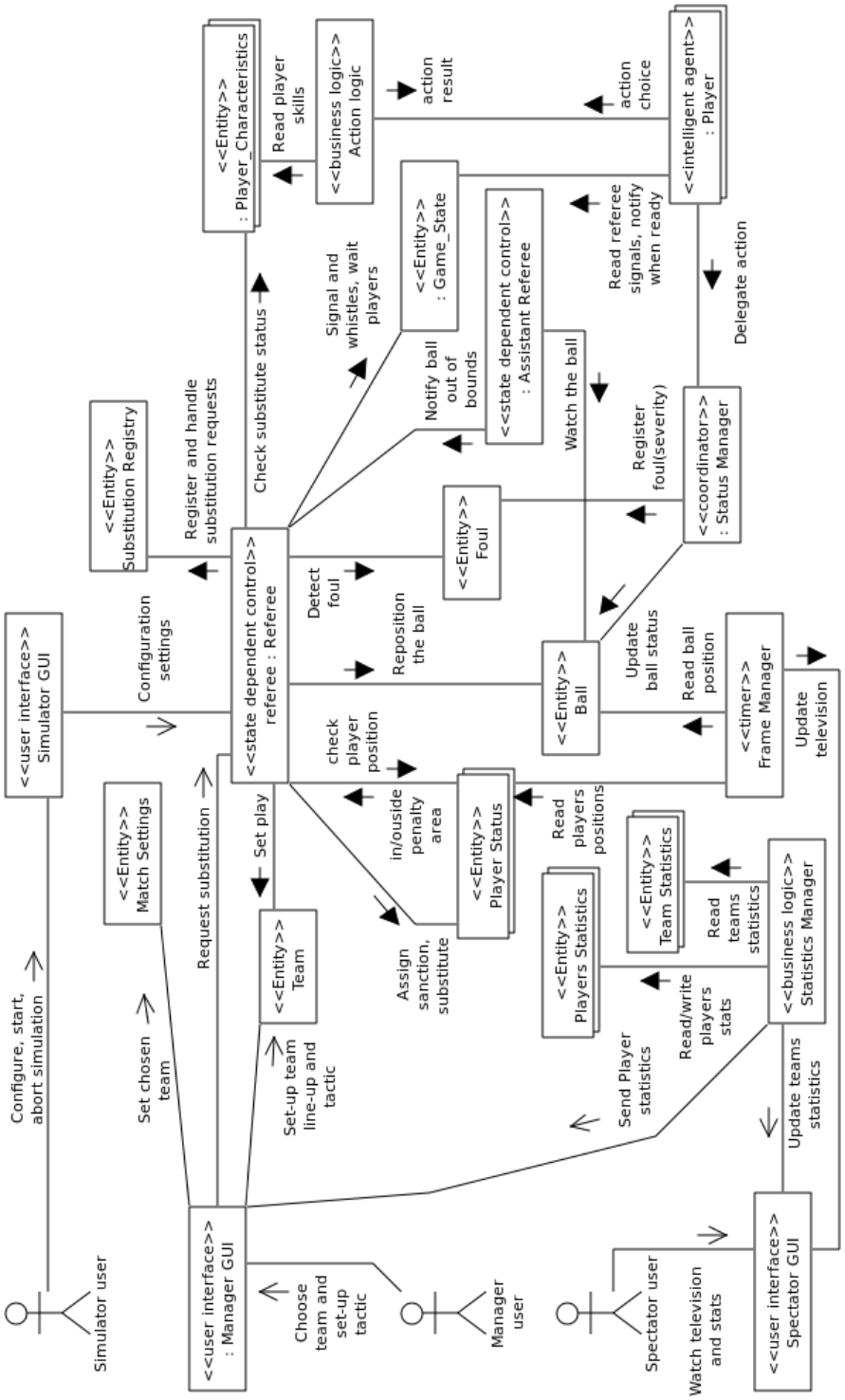

Figure 5.1: Consolidated artifacts

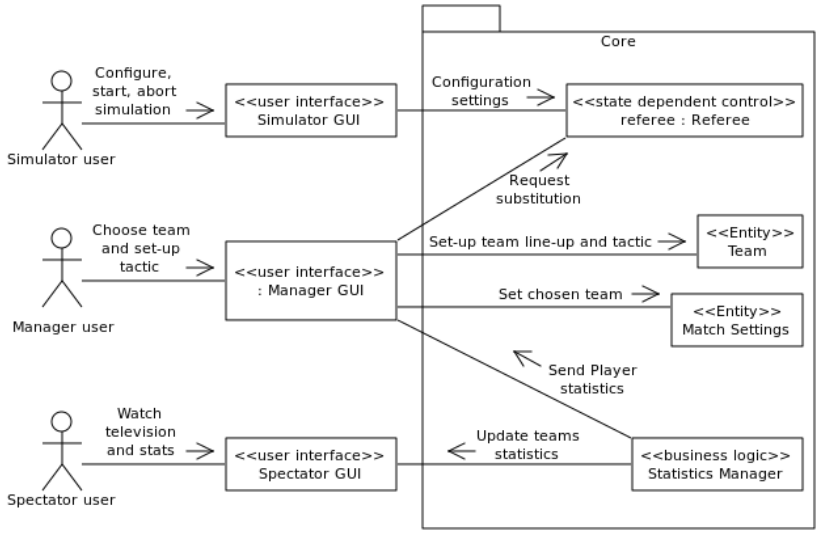

Figure 5.2: Subsystem Structure

mation but with a different update frequency). The billboard subsystem also keeps track of the number of connected interfaces. If a third Manager user attempts to connect, the validation of his interface will be rejected. The same holds if a second Simulator user attempts to connect.
The whole system, comprehensive of the billboard subsystem is depicted in Figure 5.3.

For the sake of load balancing, the core and billboard subsystems can be distributed in distinct nodes. The same can be done for users; having them in distinct nodes we also improve the overall usability of the system. More importantly, giving them the opportunity to interact from different computers is a system requirement. An example of a possible deployment of the system is illustrated in Figure 5.4.

We have up to one simulator manager, two team managers and an indefinite number of spectators. All of them could be connected from different nodes, but nobody is actually required to connect and the simulator should still work without any human intervention (except for the one who execute the Core). For this reasons we indicated zero as the minimum multiplicity of each user interface.
The deployment diagram also makes clear which nodes are actually supposed to be (potentially) distributed. A more punctual description of the distributed architecture is presented in the next chapter.

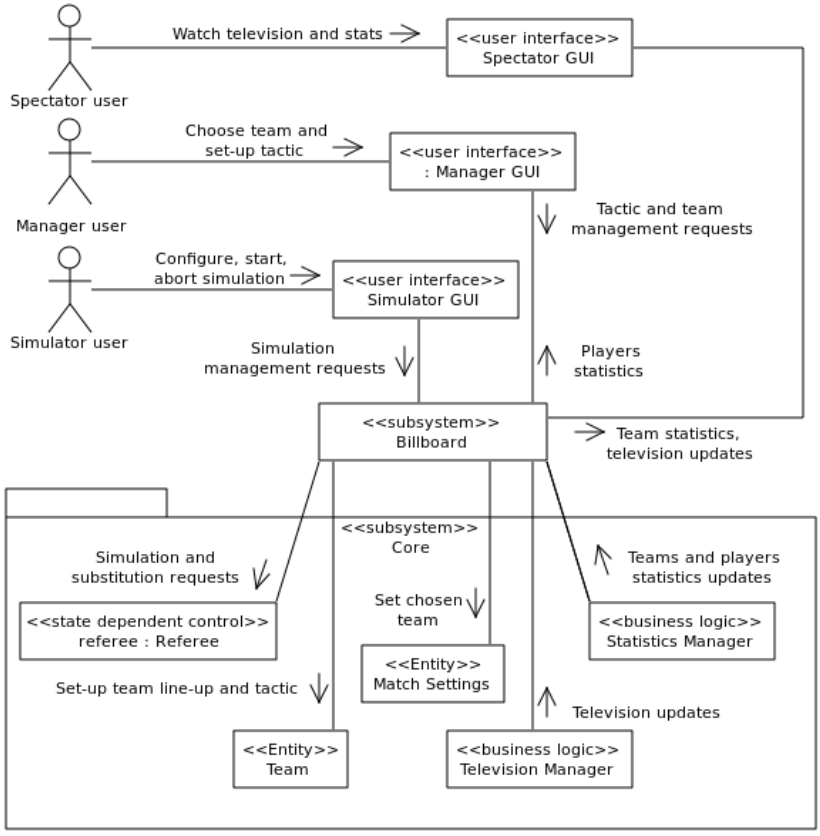

Figure 5.3: Subsystem Structure (refined)

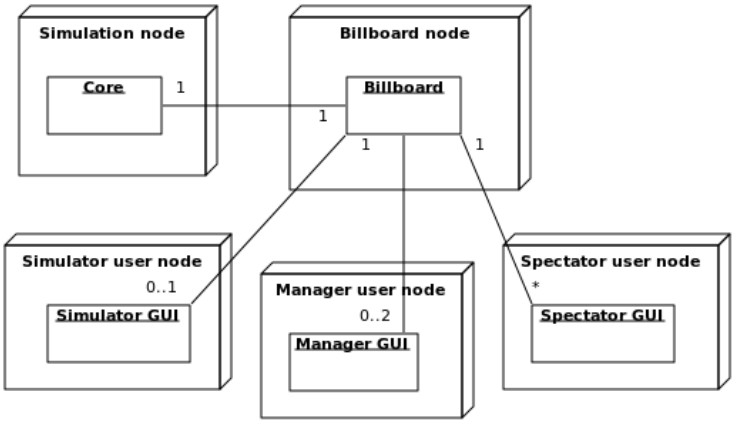

Figure 5.4: System deployment

Chapter 6

Distributed Component-Based software architecture

6.1 Reasons to construct a distributed system instead of implementing a centralized system

Our reasons for constructing a distributed system instead of implementing a centralized system are:

1. **Scalability** Denotes the ability of a system to adapts its performance flexibly to increased demands that exceed the processing power of a single computer. To reach that goal, it is necessary to partition the system into different components that reside on separate network hosts and that are able to communicate with each other.

2. **Heterogeneity** We need to build some communication mechanism between Ada and Java. The latter provides better visual interfaces and we considered easier to develop a good visual application using Java.

3. **Hiding the distribution aspects from the system users** The users perceive the distributed system as one entity and cannot distinguish it from a single, integrated computing facility. During the development, the entire team is not bothered with the complexity resulting from the system's distribution. Team members can proceed as they do during implementation of one monolithic application [11] [12].

6.2 Design distributed component-based software architecture

This simulation system, as we already mentioned on the previous chapter, is composed by different subsystems. Three of them are Ada partitions that com-

municate using Remote Call Interfaces, defined on the Ada Distribution System Annex. The other partitions are pure Java front-end programs; their main purpose is to show graphically all simulation performance and interaction with the end user.

In order to let Ada partitions communicate with Java GUIs we use the YAMI4 framework.

YAMI4 is a set of messaging libraries designed for distributed systems with particular focus on control and monitoring systems, is available for Ada, C++, Java, .NET and Python and is supported on POSIX-compliant systems, Microsoft Windows and Java-based platforms [13]. Yami4 has many features; our simulation is using a few of them:

- **simple publish-subscribe messaging**

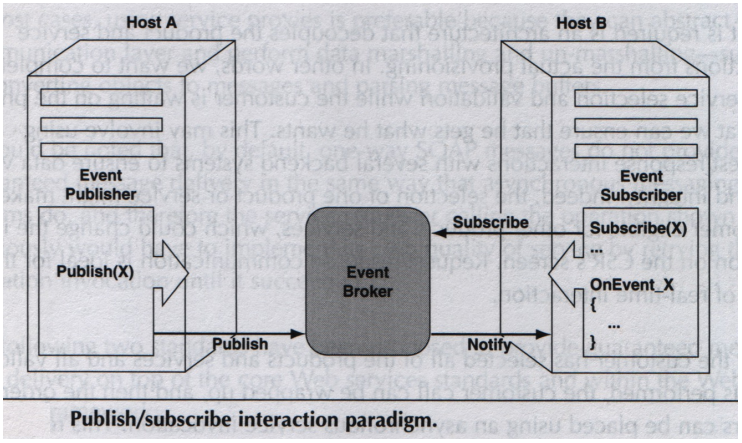

Figure 6.1: Distribution, publish-subscribe messaging [14]

- **simple client-server system**

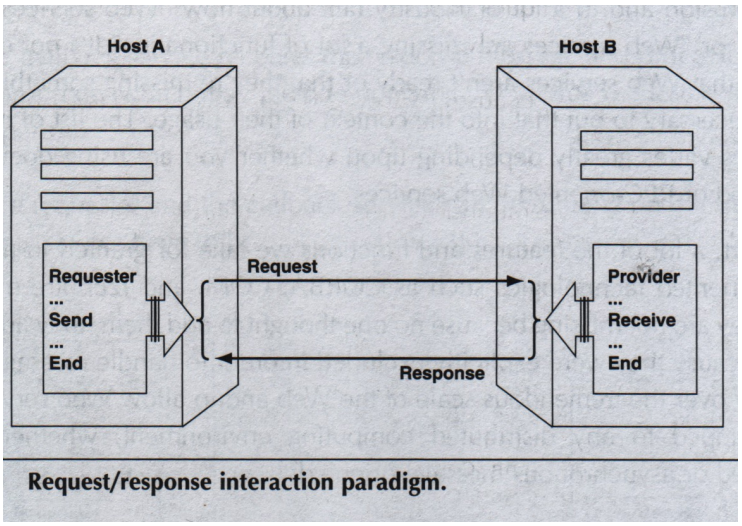

Figure 6.2: Distribution, Client-Server [14]

6.3 System configuration

There are three Ada partitions:

The **first partition**, the most important, is named **Core**; this partition executes the main concurrent simulator's business logic; this partition has a remote interface called "CoreListener", which receives all the notifications. Core receives information from the user interfaces through the "Billboard Input" subsystem and is built using Ada standard libraries.

The **second partition**, is named **"BillboardOutput"**. This partition is used to notify the activities performed by Core partition to the outside. It exposes a Remote Interface called BillboardOutput to the Ada Core subsystem and use different YAMI4 Publishers to notify the GUIs.
When the Core business logic has to notify something to the GUIs, Core simply call remote procedures defined in BillboardOutput.
When BillboardOutput receives some notification, it transforms all the Ada data parameters into YAMI4 parameters and publishes them on the correct YAMI4 Publish channel.
BillboardOutput partition has a basic state, it saves the last incoming FotogramFrame, which will be read by two tasks that sends notifications on the "RegularFotogramFrame" and "SporadicFotogramFrame" channels.

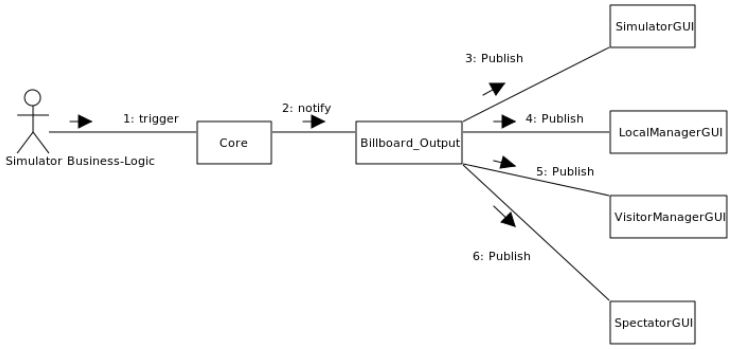

Figure 6.3: Distribution, Output Communication (PUSH-Model)

The **third partition** is named **"BillboardInput"**. This partition is used to notify the core about user requests. This partition in one side (Java) exposes a YAMI4 server, and by the other side use the Core remote interface CoreListener to notify each request.
When a user makes a request using a GUI, the client component of the GUI communicates to the Server using YAMI4.
When YAMI4 server receives the request from the BillboardInput partition, it validates the request, transforms the YAMI4 parameters into Ada parameters and then call the requested procedure located inside the Core partition.
BillboardInput partition has a little state; it contains some counters to track the current number of connected GUIs; to guarantee the consistency of these counters we use a protected type.

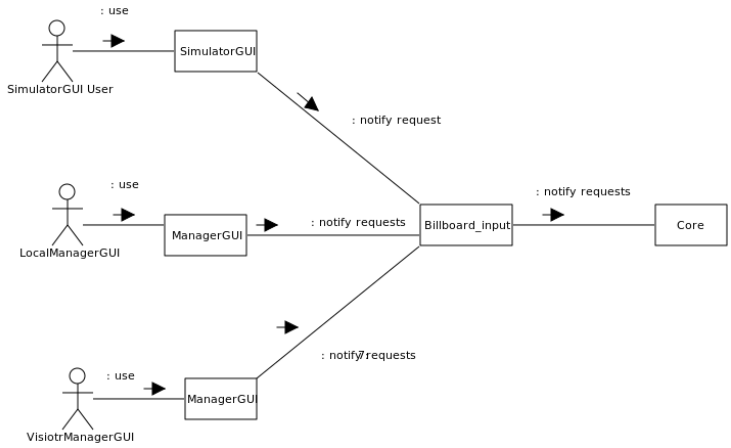

Figure 6.4: Distribution, Input Communication (PUSH-Model)

The front-end Java views are four:

1. SimulatorGUI
2. LocalManagerGUI
3. VisitorManagerGUI
4. SpectatorGUI

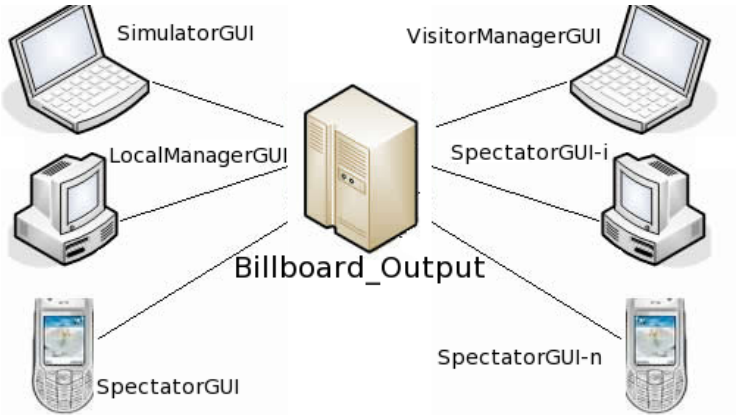

Figure 6.5: Distribution, BillboardOutput Publisher - GUI's Subscribers

Each of these GUIs implements a set of subscribers, according to the Publish-Subscribe communication paradigm.
To establish a connection, SimulatorGUI, SpectatorGUI and ManagerGUI implement a Client, according to the classic Client-Server communication paradigm.

6.4 Startup execution order

The system should be started in this specific order:

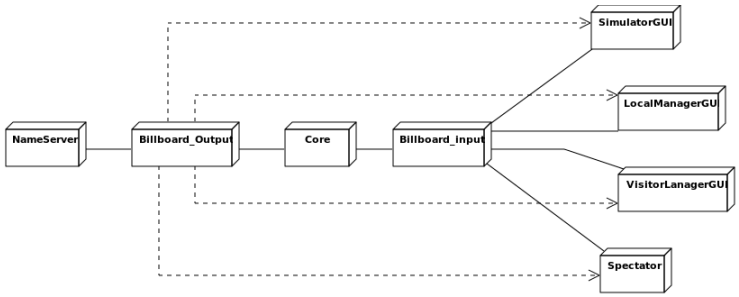

Figure 6.6: Distribution, startup execution order [15]

1. Start the name server, in order to register all DSA partitions;

2. Start BillboardOutput, which registers on the name server and makes the remote interface available. It then instantiates Publishers and registers them into YAMI4 agents; this action permits the further Java nodes' callback mechanisms;

3. Start Core, which registers on the name server, makes the remote interface available (CoreListener) and it prepares itself to use the remote interface BillboardOutput;

4. Start BillboardInput, which instantiates a server and registers it into a YAMI4 agent; registers the server remote interface BillboardInput to the name server and prepares to use the remote interfaces CoreListener and BillboardOutput used to dispatch requests. This actions permit the GUIs to make requests.
   The execution of the interfaces listed in the following points is not required and the simulation can perform also if they never run. The GUIs can be executed in any order, at any time.

5. Start SimulatorGUI, which registers itself to the BillboardOutput channels:

   - MatchStatus
   - SimulatorResponseInquiry.

6. Start LocalManagerGUI, which registers itself to these BillboardOutput channels:

   - MatchStatus
   - LocalTeamPlayersStatistics
   - LocalMessages
   - LocalResponseInquiry

7. Start VisitorManagerGUI it register to BillboardOutput channels:

   - MatchStatus
   - VisitorTeamPlayersStatistics
   - VisitorMessages

- VisitorResponseInquiry

8. Start SpectatorGUI, which registers to these BillboardOutput channels:
   - FotogramFrame
   - TeamsStatistics
   - SpectatorMessages

The GUIs registration allows them to be notified automatically through a callback when notifications arrive on these channels; all this features are provided by YAMI4.

6.5 Summary

Distribution was designed based on the PUSH model using Middleware and asynchronous communication, which hides the network delays and avoids temporal coupling.

Our design is based on events, with attention for the referential decoupling between components; it also have a little persistent data space (Billboard) that help us to enforce the temporal decoupling.

We take care of the scalability, so the system can run with or without some resources (Java interfaces). Our components can be connected at different distances, and should maintain the same performance level.

The simulation inside the core keeps running also if the other partitions fails; Core generate its persistent logs, then when the simulation ends, we know exactly what happened in the entire simulation.

We have the most important algorithms centralized on a unique partition (Core), we distribute services like connection requests and other notifications like FotogramFrame that is performed in 3 different periods named frame rates.

To ensure the syntax and semantics of the communication and proper interpretation we use Ada DSA (using polyorb name service) and Yami4. This saved us all proxy/skeleton creation, parameters marshaling/unmarshaling, binding and eventual problems with persistent objects and static/dynamic invocation.

Chapter 7

Concurrent Design of the Core subsystem

In this chapter we describe the design of the Core subsystem. In the first part we describe the aspects that relates to the simulation itself, starting with the main design choices (from 7.1 to 7.3) and then describing concurrency issues and solutions (7.4 onward).
Afterwards (7.7), we describe the generation of events and statistics along with tasks and structures used to handle data sent to other subsystems. Finally we introduce the initialization and finalization of the core subsystem.

7.1 Design of Core subsystem

The core subsystem needs many concurrent tasks to execute the simulation. Most of the tasks shape objects that are concurrent in nature, so the decision that those entities had to be modelled as tasks was just a consequence.
Players execute their own business logic, as described in Section 7.3.1. We will see that they are agents that iterate a sequence of a few steps: look around, determine an action, do the action and wait a duration coherent with the action done.
The referee is supposed to watch the game and to stop it as soon as something happens that requires him/her to stop the game. Something can be a foul committed by a player in the attempt of tackling the ball carrier or a ball falling out of bounds.
In the latter case, the event is captured by the Assistant Referee (the linesman). Because of the characteristics of the simulation, we do not actually need two linesmen. One single task is absolutely enough to check them all and avoids to overload the system with an useless task.
The business logic of the referees is explained in Sections 7.5.2 and 7.5.3, but the main idea behind it is that we do not want them to make use of polling to check what happens. First, because it is absolutely inefficient. Second, because we do not need it.
If a player makes a foul, this event gets known as soon as the foul condition is evaluated. This mean is that we can more easily notify the referee of the foul. This apply as well for the assistant referee. When the ball falls out of the field

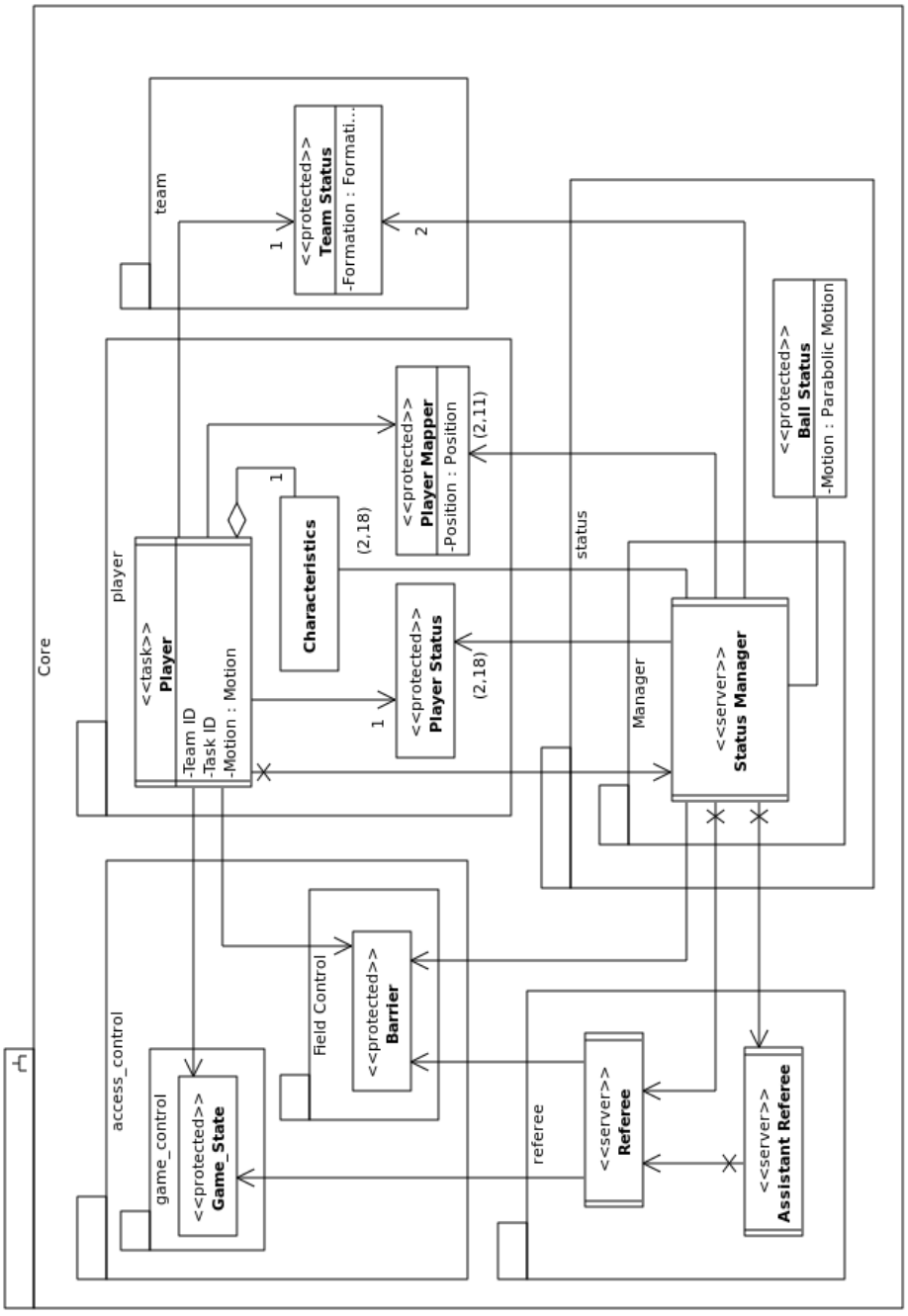

Figure 7.1: Design of Core subsystem.

area, this can just be notified to her.
The ball is represented by a passive entity (a protected object) called “Ball Status”. Conventionally, we used the suffix “status” to identify any dynamic characteristic that somewhat represents the status of an entity. The ball, being passive, can actually “move” according to some defining function, that tells where the ball is in any moment of a continuous or discrete time frame. Its motion can also be stored in advance as a sequence of steps and timings that express the function above in a - necessarily - “discrete” fashion.
The main advantage of having the ball as a passive entity is that it could move as in nature, as long as the physics of the motion is sufficiently well modelled, without any hassle due to task scheduling. The main disadvantage is that it actually adds a Real-Time constraint to the system, as we explain afterwards.
We already anticipated the need for some special entity that makes “real” the actions done by players. We called that entity “Status Manager”. It can be seen as something that updates the status of the game on behalf of the player. It’s also the entity that represents the eyes of the referee staff because it tells the referee assistant when the ball goes out of bounds and the referee when a player commits a foul.
In Figure 7.1 we present the task structure of the Core’s simulator component showing the directions of the associations (and also communications) between tasks. With that notation we would like to give some confidence about the nature of each task. None of our active tasks is also a server task at the same time (active tasks make calls but do not have entries, cfr. [4, p. 123]). The status manager is the only server task that actually makes calls (this consideration is not correct if we consider protected objects, but this aspect will be discussed afterwards), but in its calls the status manager is only delegating actions to the other referees. There is no way to have all these tasks involved in a circular wait because there is no way to obtain a circular chain of calls between these tasks.
In the same diagram we are somehow anticipating the nature of those tasks, that can be easily mapped to an Ada implementation

- “protected” for protected types/objects;
- “server” for reactive tasks that receive calls;
- “task” for active tasks that execute calls (we acknowledge that this name does not carry out much information by itself).

We believe that this use of stereotypes is acceptable because the concept we are expressing with these keywords are really general concepts, such as (active) tasks or servers.
To keep our presentation consistent along our description of the design, we are also revealing some aspects that anticipate some detail of concurrency design, such as objects that are synchronized. We used the word “protected” to make clear that those entities can be accessed by multiple objects concurrently and then require a mutually exclusive access in write mode. In the following section we will give some details of the most important aspect of the design. Some detail of the business logic is also provided when relevant. Then we will describe concurrency issues and solutions.

7.2 Modelling the field, positions and motions in a discrete fashion

The field is represented by a matrix of 105x68 cells of one meter each one; positions inside the field are represented by integer coordinates, in which X is used to indicate the position along the sideline, while Y refers to the goal line and Z is obviously the height. The unit is the meter, positions with X=0, X=106, Y=0 and Y=69 are outside of the field. Position (0,0) is outside of the top-left corner. The field is also made of lines and spots and partitioned in smaller areas. The use of these areas enhances the ease of use of positions in calculations, helping the referees to check some game condition (e.g. the fact that the ball is inside the penalty area when a foul is committed). Field and player coordinates are two-dimensional, while those that relate to the ball are three-dimensional.
Having discrete coordinates also mean that movements become jerky (otherwise we should use smaller units), but also simplifies how players see and act inside the field. We adopted the solution that only one player can be in a single position.
Player moves according to a linear motion, with the motion obviously rounded to meters. The ball has instead a parabolic motion, that is, the combination of a linear motion and a uniformly accelerated linear one. The linear motion is implemented for the ball as for the players. The Z component, that changes according to the acceleration impressed by gravity, is a floating point value (rounded in cm for the GUIs).
We simplified the model of our problem deciding that players kick the ball giving it an initial angle that is fixed (let's say, around $\pi/6$, except for the special case of a player heading the ball into the net).
Things are a bit more complicated because we wanted to have the ball bouncing inside the field until it stops. According to an experiment [7], a bouncing ball looses about 25% of its power at each bounce. We did not investigate more on how and under which conditions this applies, nor whether these data are real or not in a soccer game, because we think that the approximation is basically good enough for our purposes.
In our implementation we decided to implement the ball motion pre-calculating it every time a player kicks the ball and then storing it in an array. This, again, makes things easier because we cannot otherwise use the physical equations of a trajectory "as is". We need to approximate the linear component of the motion and we would also need to add to the equation something that models the changes of the motion during each bounce. Implementing an algorithm that pre-calculates each step of the whole motion and stores it in an array is far easier. This way, players can use one of their main cycles to calculate the motion, and the following cycles to execute each step one by one, at the right (pre-calculated) time that corresponds to the time needed by each move.
It might seem strange that the movement currently made by a player is also stored inside an array. This is actually due to the fact that the player task executes a state machine and for each loop it makes a step forward in some direction. Anticipating some detail of the player's business logic, when the player decides to reach a particular position, he would have, for each step, to decide the next position. Now, if the decision is to greedily choose the one

that makes the player as closed as possible to the destination, a step along a diagonal would always be preferred, ending up with a weird movement. To fix this problem we should probably adjust the choice weighing the distance to the target position with the time spent to make the step (higher for a diagonal). Again, calculating and approximating the linear movement a priori, storing the results inside an array, is pretty easier.
We still have some problem, indeed. How can we record the time spent to reach a position, if we do not use the real positions? In our solution, we decided to refer to the distance from the starting position: we do not calculate when the ball/player arrives 'there', but how much time it/he takes to move that much.
Another question, which might seem in appearance trivial, is "how does the ball stop?". First of all, if it goes out of bounds, we stop it immediately because afterwards a player will have to throw it in again (and we do not have any reason to show it running far away, maybe with a ball boy running after it).
But the fact is that if its movement slows down of 25% at each bounce, the ball would keep running forever, slower and slower (until it gets out of bounds). A misbehaviour would be then observed in players, because to intercept the ball they just run to the first position in which they can arrive before the ball. We are aware that the word we wrote, "before", is just a matter of design. But we can fix the problem of a player waiting a couple of seconds next to the ball for it to come there (because the ball is really slow while doing that step), along with the eternal movement not allowed in nature, just pretending the ball to stop in a reasonable way. Any, even unnatural - but reasonable - way (asking physicists for forgiveness) would work. That's why, at the end, we are just stopping the ball after it has been stationary for a while (e.g. a second seems a good trade-off).
Finally, a player who wants to stop the ball calculates whether he can reach it somewhere, at an affordable height.

7.3 Players as "intelligent" agents

Players could be defined as software model-based reflex agents.

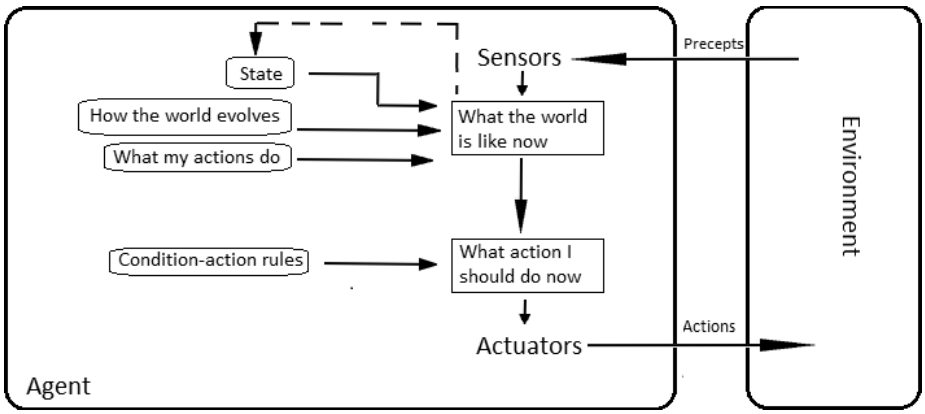

Figure 7.2: A general model-based reflex agent (source : [10]).

They are software agents because they perceive the environment just check-

ing the state of the game inside some shared data structure and they act making changes (usually indirectly) on the same structures.
They are reflex agents because they behave on the basis of some simple condition-action rules.
They are mostly working as simple reflex agents, but we define them as model-based because they actually store some information about what is going on in the system. They need at least the minimal information that prevents them from falling into some "embarrassing" cycling situation (e.g. attempting a tackle 30 times in a second because they never realise that they already tried 0.03 seconds before).
They are not goal/utility-based [10] (they don't even have the "concept" of goal, nor they have any metric to measure the convenience of choosing between different actions). There is also no machine learning technique in use, clearly. We wanted to keep them as simple as possible to focus on concurrency and distribution aspects, and still their business logic ("agent program") has become quite complicated in the aim of having them playing in an acceptable way.
In many situations, their interactions would drive them in a "composite" cycling in which two or more players keep repeating the same actions forever. These situations are not avoided by their model of the world, but they are prevented by the use of randomization. To all extents, randomization was not introduced as a workaround for solving this issues, indeed, but to add some level of non-determinism and to implement the physical characteristics of players. But the side effect that randomization gives to the players is still, absolutely welcome.

7.3.1 Player logic

Figure 7.3 illustrates the main flow of the agent program. The player task starts reading from the environment, then it chooses an action according to his business logic and calculates the effects, before doing it.
The action is delegated to the Status Manager task; the player then waits a delay strictly dependent on the time needed to perform the chosen action.
The set of condition-action rules are shown in Figure 7.4.

It is probably already clear that players execute a statechart, and the sequence of steps that drives each player in choosing an action is repeated for each main loop. We wrote that players are to some extent more similar to simplex reflex agents that to model-based agents. The fact is that the knowledge they have about the status is absolutely limited and during each step they check whether they reached the ball or not, no matter if they were actually running toward the ball or not, and no matter if during the previous step they where at 30 meters far away from the ball.
Some knowledge about the status history is indeed necessary, for example because it makes no sense to repeat continuously (in a really short time) an action that already failed, because we assume - for example - that if a player misses the ball in the attempt of stopping it, then the ball will run away. Sometimes this is also prevented by the delays that simulate the time needed by players to perform the action. Moreover, calculating how to reach a position every time the player makes a single step is useless and can lead to strange movements, as we discussed while explaining the motion.
But sometimes the player really needs to change his movements, no matter if

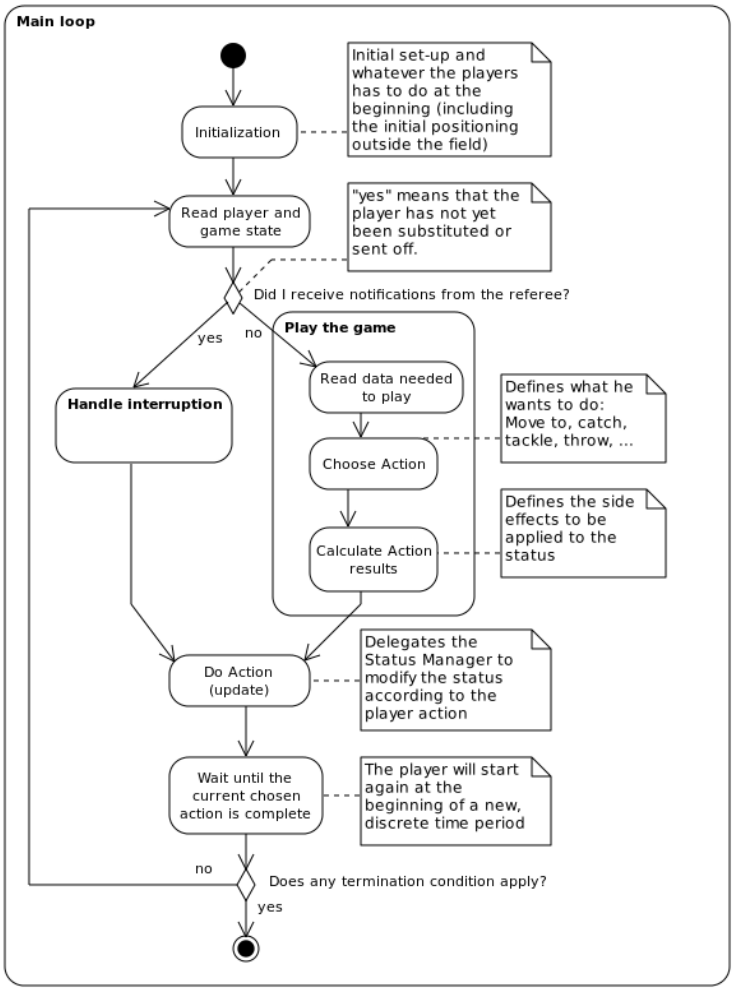

Figure 7.3: Main loop of the player IA.

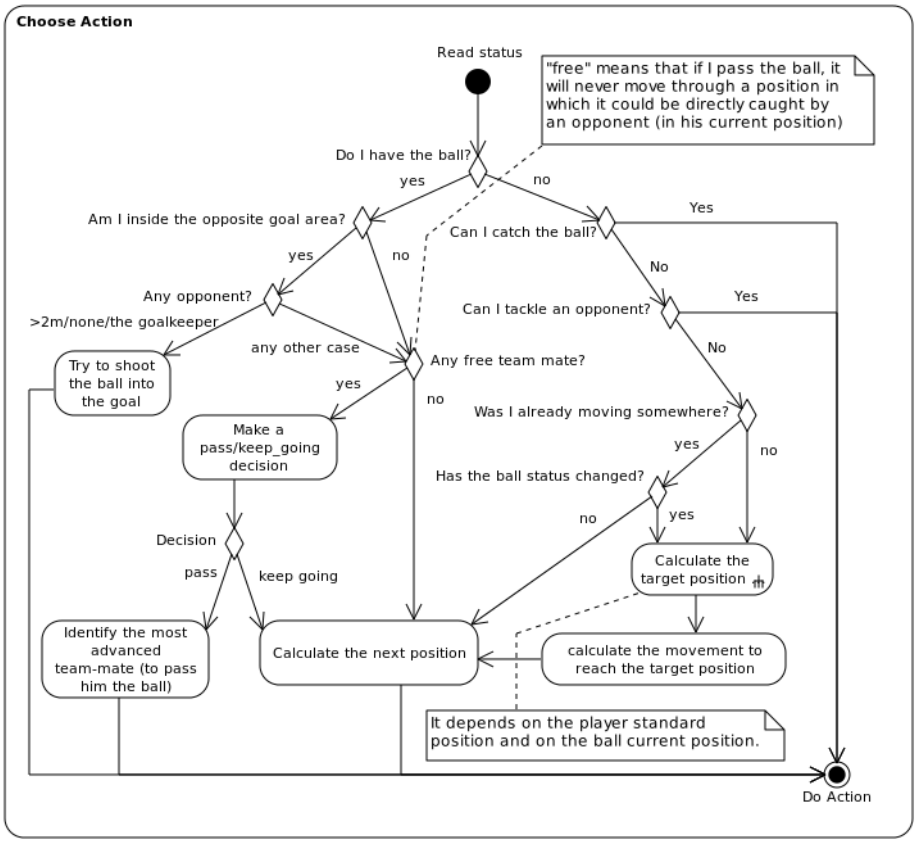

Figure 7.4: Player - Choose action.

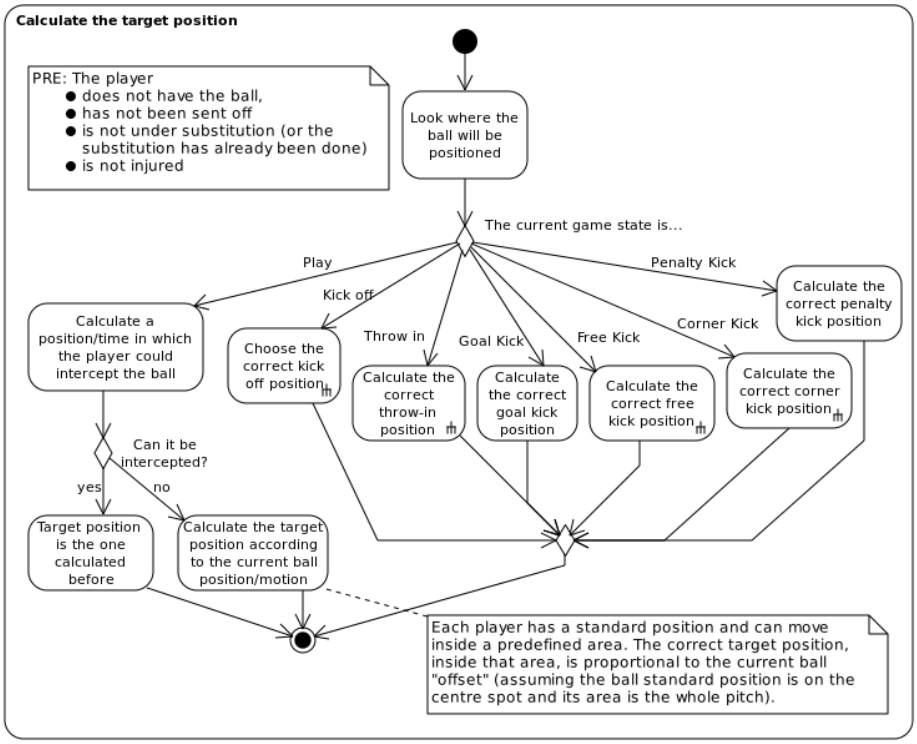

Figure 7.5: Player - Calculate target position.

the previous ones were not yet complete. The idea is that if something happens in the game status, he has to realise the new conditions and adapt his tactic to them. This happens automatically when the referee whistles, because in the next loop the player would detect and handle the interruption. But the player also "recalculates" his tactic (movement) when the status of the ball changes (e.g. somebody stops the ball or tackle the ball carrier). We found that in the other cases there is no need to recalculate the player's movement.

7.3.2 Determining player positions and movements

The meaning of calculating a movement, or defining a "target position", is introduced in Figure 7.5.

The diagram shows many cases of interruptions, recalling diagrams that we will skip in this report. What actually happens in those cases is that the player who is supposed to restart the game moves in the position in which the game will be restarted. The other players move according to some "reference" position that is practically almost always the same. In case of kick-off their position is adapted so that everybody stays in his own field side. In case of free/penalty kick they should stay at least 9 meters away from the ball, but they are not actually programmed to check this condition because their agent program does not take advantage of this (but we do not exclude that they can take advantage "involuntarily" in rare situations). The same applies for the offside rule.

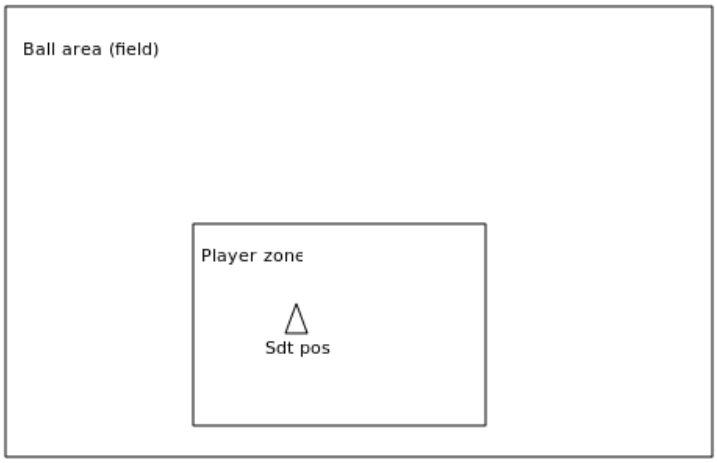

Figure 7.6: Positions.

Each formation defines, for each player:

- a "standard position", that is the predefined position associated with the player who has the corresponding role inside the formation;
- a "player zone", that is the area in which the associated player moves, when he is not the ball carrier (who is allowed to move outside his own area);

Using this information it is possible to calculate a "Reference position", that is the position in which the player should move according to the current status of the game.

The reference position of the player depends on the position of the ball when the player looks around. More specifically:

- if the ball is on one corner of the field, then the player is on the corresponding corner of his own area;
- if the ball is on the centre spot, then the player is on his standard position;
- if the ball is in any other point of the field, the player moves inside his own area proportionally.

The standard position is not necessarily in the middle of the player zone; it depends on the definition of the formation, but the important thing is that the player should move proportionally to the ball position along the field. Dynamically moving the standard position of the players along the X axis (sideline), we can change the pressing of the team.

7.3.3 Player actions

We already described how players move. The other players actions are:

- making a step forward to reach their "target position";

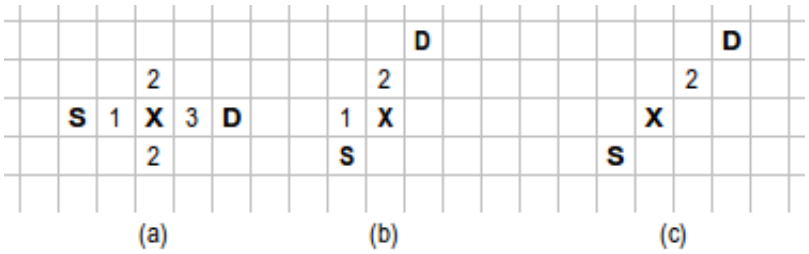

Figure 7.7: Moving from S (source) to D (destination) without clashing with player X. Note that in case (a) we have two alternatives, in case (b) we assume the original route was through the position of X and the one just below D, in case (c) the player waits and retries.

- trying to stop a moving ball (or to catch it if stationary);
- trying to tackle the ball carrier;
- kicking the ball (passing it to a team-mate or shooting it into the net).

Having these actions working is simplified by the adoption of discrete positions. We set some simple rules to determine what players are allowed to do according to their positions:

- only one player can stay in one cell (i.e. a square meter) of the field;
- a player can stop, catch, kick (pass/shot/head) or throw-in the ball if both the player and the ball are in the same cell of the field;
- a player can tackle an opponent only if he is in a neighbour cell.

7.3.4 Synchronization of player movements

From a logical perspective, because of the non-overlapping rule, making one step forward is probably (and incredibly) one of the most complicated actions. In our solution, we decided that

- when a player finds an obstacle along the way, instead of recalculating his movement (which calculation does not include any "clashing avoidance"), he looks for a neighbour cell which has a one-step distance to the subsequent one (Figure 7.7(a));
- if he can't find such a cell, he looks for any cell that is closer to his target than his current position and then recalculates the movement from the new position (Figure 7.7(b));
- finally, if he cannot find any convenient cell, he temporarily stops in his current position and then retries/recalculates his movement after a while (e.g. the time needed to make a step, see Figure 7.7(c).

About the other actions, as we have already anticipated, we use randomization to implement player characteristics. More specifically, the physical characteristics generally correspond to the probability of success/failure in player

actions. When attempting an action, we calculate a pseudo-random number somehow related to the range of values permitted by the player characteristics and we use the characteristic itself as a success/fail threshold:

- if the random value is in the "success" side of the range, the action will actually be carried out;
- if the random value is in the "failure" side, then the action will not be made but the attempt will eventually be reported as a failed attempt in the match history.

7.4 Players as concurrent tasks

In the previous section we described players as agents to introduce their business logic. We can also see players as state machines and this perspective is more useful to move toward the description of their interactions with other concurrent tasks, that usually cause state changes. In Figure 7.8 we provide a more exhaustive overview of all the possible player states. The top-left corner corresponds to the main loop we described in Figure 7.3. We have much more possible states than those described so far, and the transitions to the others happen in case of fouls and interruptions.
Players execute their main loop for each basic action (calculate movement, stop/catch, throw, tackle, step forward); handling player conditions and game states also involve many main loops.

Starting from the parts of the player behaviour that we already described above, the first issue that comes to mind is that checking the availability of a cell could lead to race conditions. Players are concurrent tasks, if they concurrently check a position and then move into the cell, they could both get inside. This violates the non-clashing/overlapping rule.
To solve this problem, we decided to implement a protocol that impose to nearby players to act sequentially, as we describe in the next section. In this aim, the way we modelled positions and player actions is extremely helpful in making this protocol logically simple.

7.4.1 Synchronization of player movements

For ease of exposition we will call "player neighbourhood" the set of cells that can be reached by a player in one step, as depicted in Figure neighbourhoods. Because players can potentially move concurrently, some race condition can occur within them if they have overlapping neighbourhoods. Preventing players with overlapping neighbourhoods from executing/moving concurrently is enough to avoid these race conditions.
By requiring to avoid concurrent movements we mean that the action of changing position should be sequential for nearby players. Avoiding concurrent execution is something stricter, because it means that if two players are nearby, they cannot do practically anything concurrently.
The latter option might seems too restrictive, but it is easier to implement and probably convenient because:

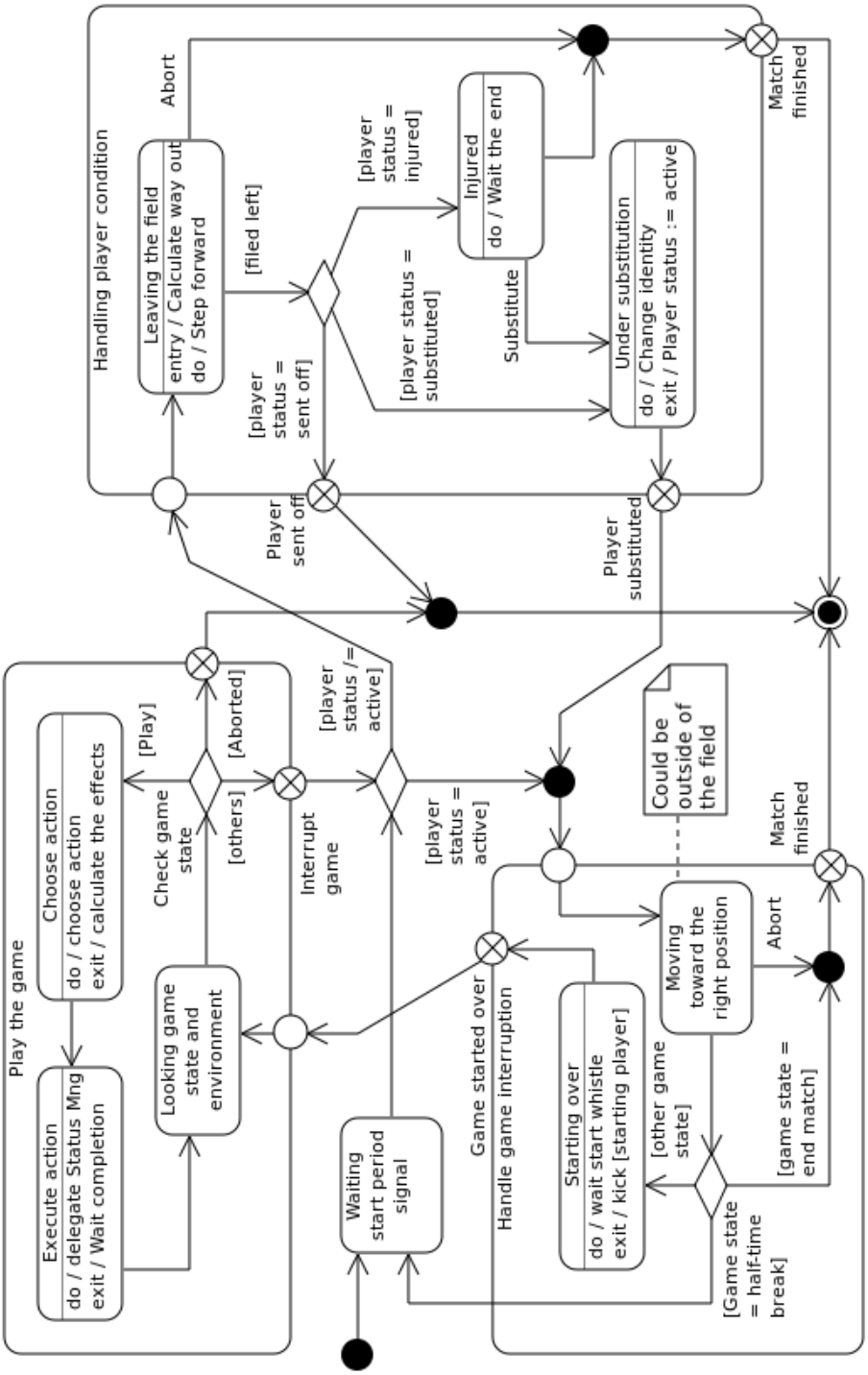

Figure 7.8: Players seen as state machines.

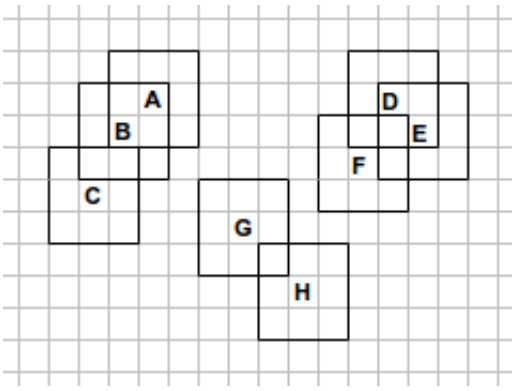

Figure 7.9: Players with overlapping neighbourhoods must execute sequentially.

- analysing the problem domain, only a few players are usually so close to each other;
- concurrency gives us the potential for gaining a higher level of parallelism, but we cannot expect an extreme level of parallelism because of hardware constraints (the level of parallelism permitted by the underlying multicore/multiprocessor system) and data dependencies (the need to synchronize threads that need to share data with each other).

In short, the number of non-nearby players is probably so high that we should not expect to exploit the full potential of their parallel execution on a standard personal computer.
A possible concurrent (and parallel) execution of players positioned as in Figure 7.9, assuming they all become ready at the same time, is shown in Figure 7.10.

The number of flows of execution that can be followed by a player is pretty high, and that is where the ease of sequential execution comes. If we want players to watch around, choose a destination and move with the guarantee that they will succeed (as long as they found it free, obviously), we need to lock their neighbourhood before they watch around. It can happen that players choose a different action: in that case the lock of the neighbourhood is a waste of others' time. But granting the neighbourhood lock only when really needed means that we need some backup mechanism to care about situations in which the chosen position has been taken by someone else after being read by the player.
Some optimization can be done anyway. For example, in case a player decides to recalculate his movement, without interacting with the outside world, he can release his neighbourhood before doing computations (note that the precalculated movement do not consider which positions are available along the way).

Here is the specification of the protected object that implements our access protocol to the field:

```
-- a protected object used as access protocol to the game field
protected Barrier is
```

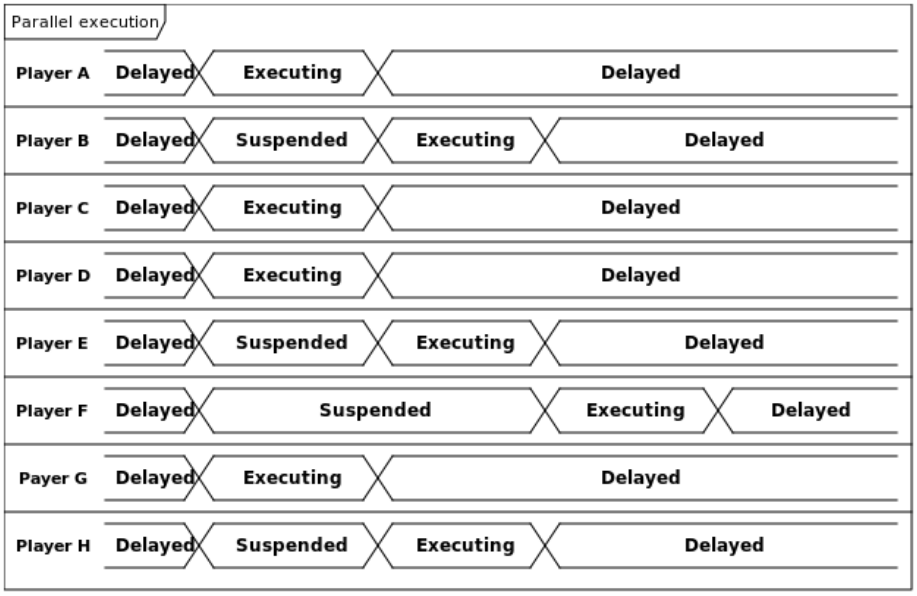

Figure 7.10: Example of a possible execution of players positioned as in Figure 7.9, assuming they start executing at the same time. "Suspended" means that tasks are waiting on some entry because they must not execute concurrently. All the tasks that are not required to execute sequentially are shown in parallel. Transition through "Ready" is not shown.

```
      -- an entry used by players to block their neighbourhood
      entry Play(Player_Position : in Coordinates_T);

      -- Unlock the neighbourhood of a player whose action
      -- is complete
      procedure Leave(Player_Position : in Coordinates_T);

      private
      -- players who need to play in a (partially)
      -- blocked neighbourhood are requeued here
      entry Awaiting_Players(Player_Position : in Coordinates_T);

      -- every time a neighbourhood is freed this variable
      -- becomes true and opens the guard of Awaiting_Players
      Someone_Finished : Boolean := False;

      -- the number of players standing in line waiting to play; when
      -- a player leaves the field, they could all be allowed to play
      -- if their neighbourhoods overlap the one which has been left
      -- (and they do not overlap each other)
      Potentially_Allowable : Natural := 0;

      -- list of players who are currently allowed to play
      Allowed : Positions_List.List;

      -- No of players registered in the Allowed list
      Playing : Natural := 0;
   end Barrier;
```

Before proceeding, we would like to remind that player tasks execute a loop, as shown in Figure 7.3. They start looking the environment: signals made by the referee (game state), position of the ball and of the other players. Then they choose and do an action, which is actually delegated to the Status Manager, and they sleep until the time expected to do that action is up. If the action is the calculation of a movement, they wait the time needed by the first step (and then they do it during the next loop); if the action is a step, then they will wait the time needed for the next one, so that they reach a position only after the time they need to reach it has passed.

"Play" and "Leave"

Our protocol requires that the player shows up to the barrier when he starts executing (before looking the environment), and notifies that he finished when he is done. The barrier then takes care of synchronizing the player with the others, eventually suspending him if some other player is executing nearby.
When players want to play, they issue an entry call on "Play"; when they finish, they call "Leave" (the call is actually issued by others on their behalf, as we will see). From now on, while writing about the protocol, we will simply say that a player entered/accessed the field to mean that he issued a call to "Play" and that he left the field to mean that he called "Leave". It will be clear from the context, outside this scope, when the same sentence will mean that a player is "really" leaving the field.
The body of "Play" checks if there is anybody in his neighbourhood who has already been granted access to an overlapping area. If nobody is playing nearby, the protected action completes and the player can proceed. If there is another player who is playing nearby, the calling player is requeued on Awaiting_Players. "Playing nearby" means that one player can reach the other in one step. If their distance is at least two, then they will both be allowed to play concurrently because they will not interfere each other (they can not make more than one step at a time).
The protected entry Play has a "when True" guard, meaning that as long as the ongoing protected action is complete, the entry can be accessed. To determine whether the player can be allowed to play or not, we need to check the parameter of the call, that is, the current player position. This is the reason why we need to use the requeue statement. Referring to this check we will just say that we are "evaluating" a player.

"Awaiting_Players"

Now the question is: how can we release players who have been requeued in Awaiting_Players? The guard opens when "Someone_ Finished" evaluates to True, that is, after the execution of a player action has finished. So, at the beginning Someone_Finished is false; the first player will not be requeued because nobody can have been granted access to the field. Other players could be requeued but if they are, there must be someone who caused them to be prevented from playing. Once the first player who accessed the field leaves, some of the requeued players will be allowed to play. Not all of them, because once a player is allowed, the following ones could attempt to play in the same area. In that case they are requeued again on the same entry. That is why we are using

a natural variable called "Potentially_Allowable": to prevent players who were already requeued from being checked again and again if nobody left since the last time they were checked.
More precisely, every time a player leaves the field and the guard is opened, "Potentially_Allowable" is set to the total number of players waiting on "Awaiting_Players"; the number is decreased each time a player is evaluated, until it becomes zero. At that time the guard "Someone_Finished" is set to False and nobody will be re-evaluated unless someone playing will leave.

Unfairness

There is a special condition that leads to an (apparently) unfair situation. Imagine that player A, B and C are positioned as in Figure 7.9. A enters the field first, then B attempts to play but he is requeued because he is close to the position of A. Then C passes the barrier because his neighbourhood do not overlaps with that of A. At that point, A leaves and B is re-evaluated. Unfortunately, his neighbourhood also overlaps with the one of C, so he is requeued again. The unfairness is the fact that C arrived after B but still prevents him from playing. This situation is illustrated in figure 7.11. We could solve this issue by using two lists, one for the "Allowed" players and one for those that were held. But this solution would be more complicated and less efficient. Moreover, should we really hold a player because he could eventually prolong the wait of a player who is already held (without any chance to run immediately)? More importantly, the unfairness is only apparent if we consider that player tasks A, B and C become ready at the same time; this happens because of a design choice that we describe in the next section.

7.4.2 Time synchronization of players with the ball

Having the ball as a passive entity with a pre-calculated motion has some implications:

- unlike players, the ball moves exactly when it is supposed to move (with no delays due to execution times and/or scheduling); this is a good thing, obviously, because once the ball is expected to be in a certain position, then it will be found right there. No hassles due to delayed active tasks;
- if a player - who would like to stop the ball - executes late, in the meantime the ball will go beyond him. Then we should try to make sure that when players wake up, they have enough time to interact with the ball, at least if they are in the right position to do so.

In our solution we decided to split time into discrete periods and to let the ball move only between different periods. This way, if a player executes inside a period (quantum), as long as he completes his action inside the same period, the ball will not move away. We also decided that players that want to make actions wake up at the beginning of a period. If many players act during the same quantum, they will all be ready at the same time and the order in which they execute is consequently up to the scheduler.
We also decided that the period should be short enough to allow the ball move at any speed and (hopefully) long enough to allow the execution of all players

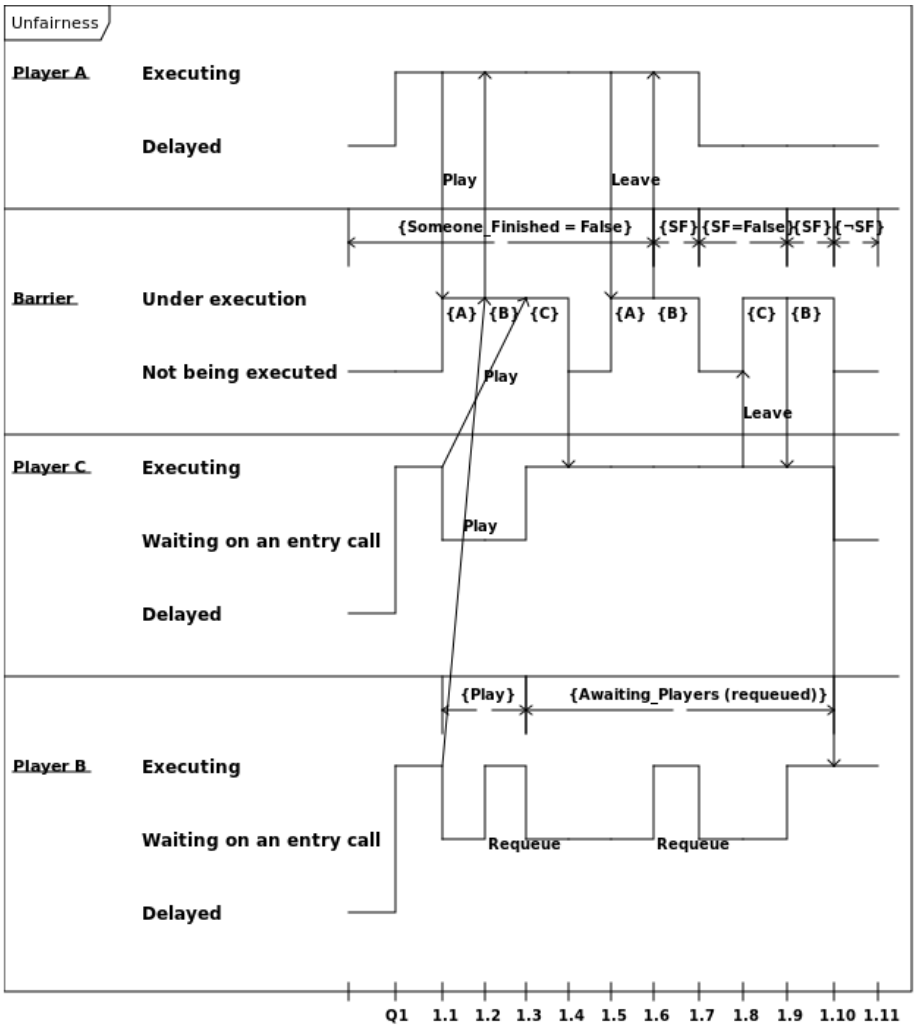

Figure 7.11: Player B is prevented from playing by player C, whose request was received later. The unneeded states are not shown. Assuming that players tasks can actually run in parallel, the transition through the "Ready" state is not shown.

during the same quantum. We found that the maximum speed ever gained by a ball during a penalty kick was less than 100-120 km/h, then we divided time into periods of 30 milliseconds (allowing a speed of about 33 m/s).
During one single quantum each player should be able to execute one time. No multiple executions are allowed during the same quantum, nor they are required to execute once for each period. When player executes they calculate the time that their action will need and then wait until that time is over. Their delay is approximated to match its expiry with the beginning of a new period.

Let's make an example. When a player decides to move toward some destination, he calculates his motion along with the time needed to reach each position. Each time is finally rounded to the beginning of its containing period. At that point, before making any step, the player waits the time slot in which he is supposed to reach that position, so that his actual arrival time, to each position, falls inside the pre-calculated period (the exact time depends on when the status manager will execute his request).
This solution allow to tune intermediate speeds very well but also imposes a Real-Time constraint, because of possible misbehaviours due to the fact that players could not complete their actions in the right time (i.e. inside the quantum).
The "Barrier", as described in the previous section, was slightly simplified because we also use it to record the time needed by all the players that decide to execute during the same time slot. We measure that duration from the first time that a player enters ("Play") to the time that the last player exits ("Leave"). If "Play" is executed in a new time slot while not all the previous players have yet left, this means that execution exceeded the time slot. Once all players finished and nobody is actually playing we consider the current composite action complete. If two or more consecutive periods are exceeded, than the execution time we calculate for the "composite action" is the sum of all the executions done on those periods.
We actually collect two information: the measured worst case execution time and the average time of all the composite actions.

Executing overtime

During tests we found that the quantum is seldom exceeded, but we have not been able to detect any misbehaviour. To our best knowledge, there is only one class of situations that might lead to a misbehaviour:

1. a player with good kick power shoots the ball with a very high speed (e.g. 30 m/s, the maximum allowed by our simulator), so that the ball moves at each quantum; it is unlikely that this situation could happen with a normal passage because speeds in that case are lower;

2. while moving, the ball stays stationary on the position of another player - let's say, the goalkeeper - for one quantum only;

3. in that quantum, a number of players execute and their execution is such that the computations they do exceed the quantum;

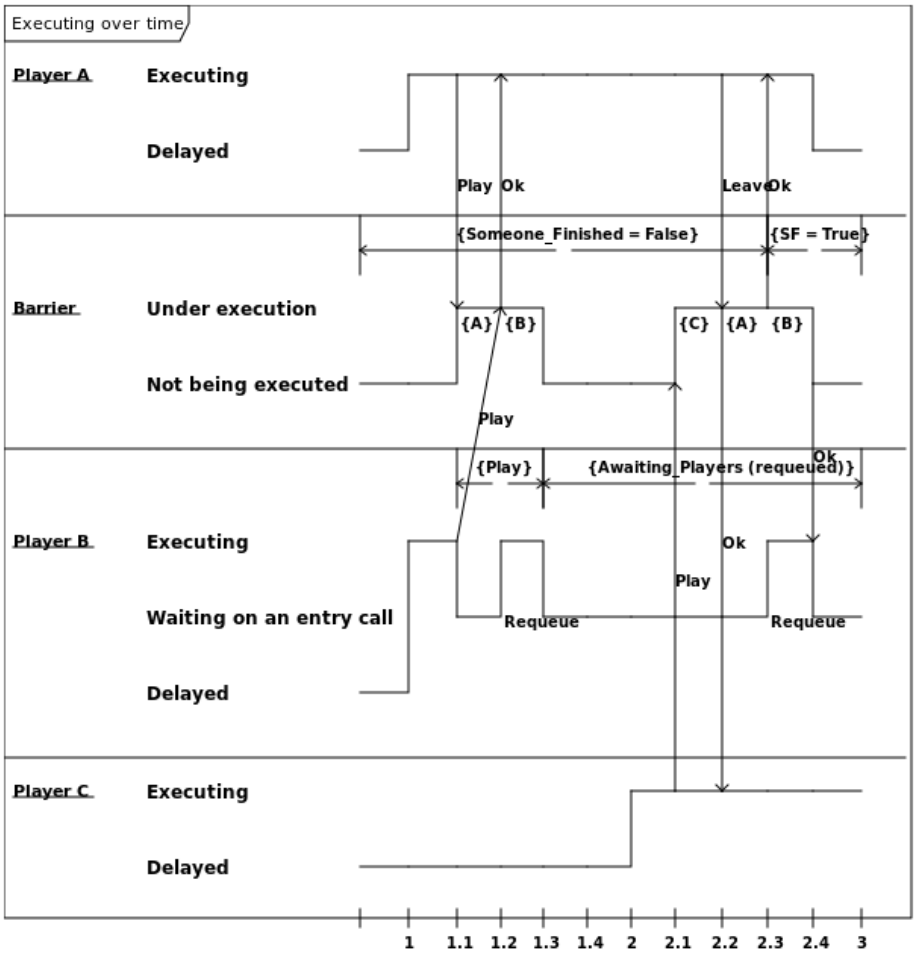

Figure 7.12: Players execution exceeding the time slot. The beginning of time periods is identified by integer numbers (1, 2, 3). The unneeded states are not shown. We assume that player tasks can actually run in parallel, then transition through the "Ready" state is not shown.

4. the goalkeeper, due to task scheduling, executes after most of the other players, so that his execution starts in the new time period;

5. consequently the goalkeeper, unlike his expectations, becomes ready when the ball has already gone beyond him. Due to the high speed of the ball, he can not follow and stop it.

Similarly, if the player sees the ball in his own position, maybe he will not end his action in time. So it could happen that the ball virtually moves away (note that the ball is passive, so it does not really move). Then the player, unaware of that, stops the ball which magically comes back to his position (because of the new setting of the ball motion). This ends up in a temporary inconsistency, but nothing else can go wrong, at least if the next ball position is not out of bounds. In fact, a player who is already positioned in the next ball position will not be able to catch it in the meantime because the access protocol is preventing him from executing. Moreover, with some probability and luck, the new temporary ball position will not even be displayed in time.

Dealing with Murphy (while executing overtime)

We can see the goalkeeper above as the Player B in Figure 7.12, which is probably the worst situation that could happen, because it brings together the "unfairness" of the previous section with exceeded time periods. Imagine that A and C are forwards playing close to the goalkeeper. One player shoots the ball. At some point A wakes up and start thinking how to cheat the goalkeeper (B). He takes so much time that he exceeds the time period. Then C becomes ready and start executing at the beginning of the new period. He sees that the ball moved on his position and understands that it will not fall into the net, so he shoots the ball again. A stop thinking and does nothing. When A finishes executing, B is requeued again because of C. As soon as C finishes, B executes but the ball is not there anymore. The hoax is that A succeeded in cheating the goalkeeper, just without doing so (i.e. doing nothing).
Now, player agents are not programmed to "cheat", what we really mean is that A needed too much time to play. Note that, despite the figure, A could finish so late because he started his execution close to the end of the period, being "ready" for a while but not executing because others were assigned the CPU(s) before. We could argue that this condition is impossible due to the logic behind player positions, but this claim relies only on the business logic and it seems quite difficult to prove that it is impossible to design a formation that could lead to this misbehaviour.
We decided to ignore this situation because it is really unlikely and - more importantly - because we set as precondition that all the players should be able to execute once in a quantum. If our precondition is not met, we cannot provide any guarantee on the reliability of the system.

7.5 The status management and the referee staff

7.5.1 Status Manager

The status manager task is delegated by the players to make the updates to the match status on their behalf. It is a server that accept calls on its entries Move,

Tackle, Catch, Throw. The meaning of these entries is quite obvious. For the latter we used the name "Throw" because it refers to kicks (shots and passes) but also to throw-ins (done by hands).
The idea behind the status manager is that players do not need to deal with what happens in consequence of their actions.
The behaviour of the status manager task is as follows:

- "Move" is called when a player changes position. This request can always be accepted because the protocol guarantees the consistency of the action. The status manager than update the player position (and the ball position, if carried by the player) and notifies the change to the task that provides the game snapshots to the outside world.

- "Tackle" is called either when successful and when it fails. When successful, the status manager updates the ball position to move it onto the player's feet. If the tackle harmed a player, it updates the status of the injured player. For the sake of simplicity, we decided that a failing tackle never harms a player. Any offence made during a tackle is always signalled to the referee.

- "Throw" updates the ball status setting it free and registering its motion. If the ball will eventually fall out of bounds, it will stops as soon as it crosses the bounds. The status manager checks if the position in which the ball is supposed to stop is out of bounds and then notifies the assistant referee of this condition.

- "Catch" updates the ball status setting it stationary on the player's feet and eventually notifies the assistant referee that the ball will not fall out of bounds anymore. If the ball was already stationary it is also updated to register who is its ball carrier and that it is not free.

Anything "relevant" causes the generation of an event that notifies what happened to the outside world (e.g. a successful/failing tackle, a foul or injury, a player kicking/stopping the ball).
The signals sent to the referees are an optimization that make them more efficient, as we will see in the next section.
When the action requested by a player is complete, the status manager invokes the entry "Leave" on his behalf. If the action is a foul, than the entry "Leave" will be invoked directly by the referee. This prevents nearby players from interacting in a way that lead to situations that would become difficult to deal with, such as multiple fouls one after the other.

7.5.2 Assistant Referee

The assistant referee receives signals from the status managers that contain information about when the ball falls out of bounds. Then he waits until that happens and notifies the referee about the place in which the ball went out (and than how the game should be restarted).
Depending on the ongoing action, the status manager calls:

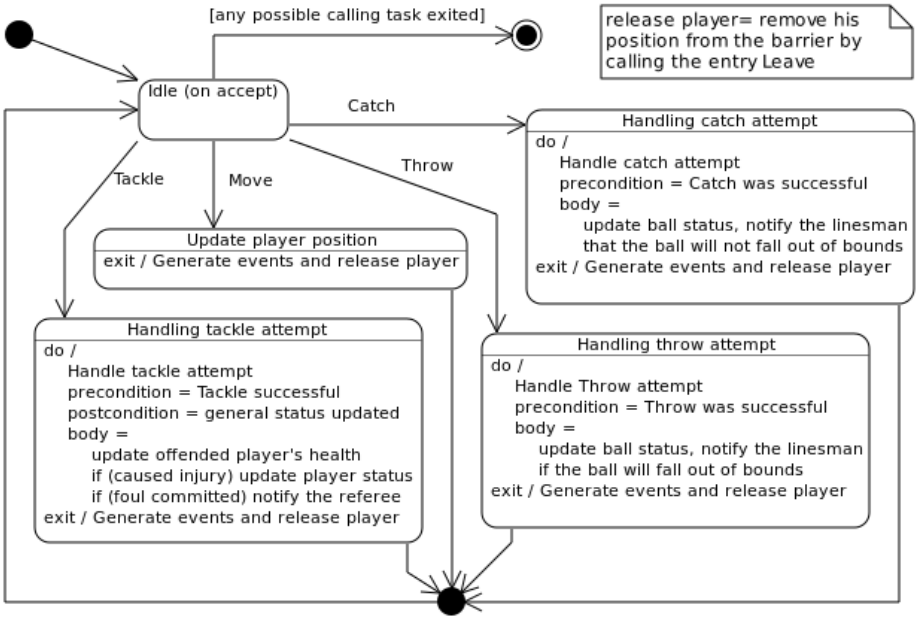

Figure 7.13: States of the Status Manager task.

- "Set_Ball_Out_Time" when, handling a "Throw" action, it realize that the ball will fall out of bounds; the expiry of the ball out time causes the assistant referee to send a notification to the referee;
- "Reset_Ball_Out_Time" when, handling a "Catch" action, it changes the motion of the ball making it stationary.

The notifications received from the status manager make the assistant referee easier and more efficient. She does not need to check the ball position each time to see whether it is still inside the field or not. Polling can be avoided because knowing in advance the exact time in which the ball will fall out of bounds, the assistant referee can just wait until it happens and then proceed with the appropriate actions.
When the ball out time expires, the assistant referee checks the current ball position and then notifies the referee about whether the interruption will be due to a goal/autogoal, throw-in, goal kick or corner kick.
The assistant referee also has a "Quit" entry used by the referee to notify him that the game is over and so he can leave (i.e. the task can terminate).

7.5.3 Referee

The referee receives notifications of fouls from the status manager and signals, from his assistant, that the ball has fallen out of bounds.
He also notifies players about interruptions of the game, any possible caution/sending off due to an offence and he checks for the expiry of the periods of play and of the duration of the half-time break.
At the beginning of the game, the referee signals that the game is going to

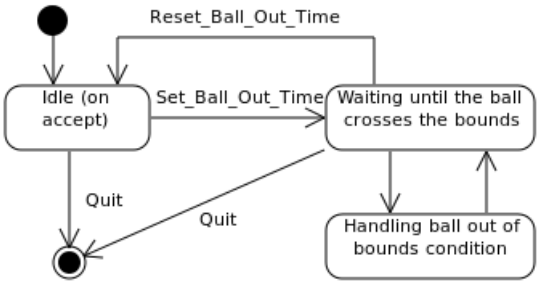

Figure 7.14: States of the assistant referee task.

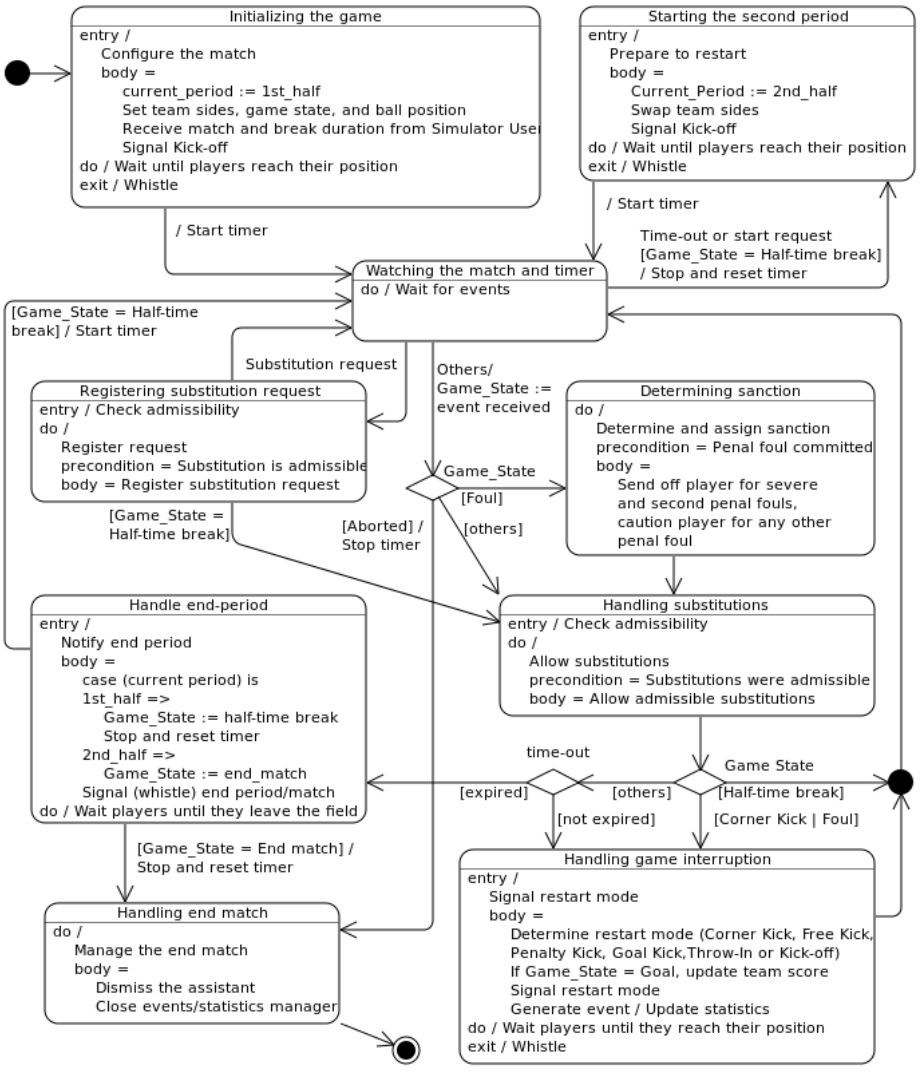

Figure 7.15: States of the referee task.

start with the first kick-off, so that players start positioning. After the kick-off whistle, he sets a time-out at the end of the first half and start accepting any interruption that may come from the status manager, from his assistant or from the team managers.
If the referee receives a substitution request during the game, he registers the request. At the first interruption he checks if the request can be accepted, according to the laws of the game, and then he handle the substitution.
When the first period is over, he waits until the first interruption before whistling the half-time break and changing the side of the field associated with each team. Substitutions requested during the break are handled immediately.
At the end of the break he tells players to position themselves for the start of the second half and - as soon as they are ready - he whistles the kick-off. The second half is obviously managed as it was for the first. From the need to accept requests from outside it is clear that the referee is a server. Receiving notifications is obviously clever than polling the game state for anything that might occur. The need to check for time-outs results in having an "or delay" alternative as it happened with his assistant.
The behaviour of the referee is more linear than that of the players, but there are many possible pitfalls because the "linear" logic of the referee must take care of everything that might happen at any point of the game, still applying correctly the laws of the game.
Figure 7.19 shows the states of the referee, also showing how the game state changes according to events received by the referee. In a transition that starts from "Watching the match and timer", we wrote wrote "others/ Game_State := event received" to mean that in any case not explicitly written in other transitions that start from the same source, Game_State is assigned a value that generally corresponds to the event notified to the referee (the only exception is that Game_State becomes Kick-off when the event received is "Goal"). That value should be considered as internal to the Referee task. During our description we wrote about notifications sent to the players, but we did not write about how they are actually sent. As long as the status manager and referees are servers, the communications described so far are done via rendez-vous. But players are active tasks instead, so in order to avoid calling them we decided to adopt a protected object that also acts as a protocol.
That protected object, described in the next section, contains a "Game_State" variable similar to the one discussed above.

7.6 From players to referees: a matter of synchronization

During the design of the core subsystem we decided to develop players as active tasks and to avoid making calls to them.
But it happens that they need to receive some notification, for example when they get injured or they are sent off, and also when the game gets interrupted for any reason.
We designed a mechanism to let them check what is going on before making any decision about possible actions, and to behave consistently with the game. As it happened with moves inside the field, we developed a protected object that -

along with the Barrier - implements a protocol between players and other tasks.

7.6.1 Managing the game state

In this section we focus on how the Game_State is managed; in other words, when the referee decides to interrupt the game for any reason, the player must be aware of what is going on and do the right things to respond to the referee signal.
We developed a protected object that contains the game state and allow synchronization between the referee and the players. In the listing below, for simplicity, we describe the specification of that object along with the guards that determine whether access can be granted to a caller task. A few procedures have been omitted because not necessary for the purpose of this explanation.

```
-- a protected resource to store the game state
protected Game_State is
   -- register the signal given by the referee while whistling
   procedure Set (New_State : in State_Record_T);
   -- let players check what happened
   function Get return State_Record_T;
   -- used by the referee to restart the game (whistle)
   procedure Start;

   -- Increase the number of players the referee should wait
   -- before restarting the game
   procedure Increase_Players_Number;
   -- Decrease the number of players the referee should wait
   -- before restarting the game
   procedure Decrease_Players_Number;

   -- used by players to wait the restart whistle
   entry Wait_For_Restart when State_Record.State = Play;
   -- to wait players until they enter the field
   entry Wait_For_Positioning when State_Record.State = Kick_Off
         or State_Record.State = End_Match;
   -- used by the referee to wait players before whistling
   entry Wait_For_Players
         when Wait_For_Restart'Count = Players_Number;
   -- wait until players leave the field before accepting
   -- an anticipated end of the half-time break
   entry Enable_Start_2nd_Time_Request
         when Wait_For_Positioning'Count = Players_Number;
   -- Used by anyone who need to wait until the game is over
   entry Wait_The_End when Players_Number = 0;
private
   State_Record   : State_Record_T;
   Players_Number     : Natural range 0..22 := 22;
end Game_State;
```

The procedures to set and get the game state practically corresponds to the actions of notifying the reason of an interruption (referee) and watching what happened (players). “Start” can be seen as the whistle itself.
When the referee changes the game state, players are not immediately aware of what happened. If they were already doing an action (tackle, kick, move, ...), they finish doing it and they check (“get”) the game state just afterwards (at the beginning of a new cycle of their main loop). If they were not yet executing, they will see the game state already updated. Here the field access protocol really helps, because it prevents consecutive fouls and other complex situations

from happening.
The protocol actually have some synchronization mechanism, realized by means of the entries of the protected object.
During the warm-up and half-time break periods, players are outside of the field waiting to get noticed by the referee that the new period of play is going to start. This is made possible by the use of the entry "Wait_For_Positioning" that opens when the referee sets the game state to Kick_Off (this condition is true at the beginning of the two periods and after each goal). The game state is obviously set differently during the non-playing phases.
After having set the game state to Kick_Off, the referee waits until all players are positioned. This is possible by issuing a call on "Wait_For_Players". Each player moves to the position in which he is expected to be to let the game start, then wait until all the others are done, by calling "Wait_For_Restart".
The Wait_For_Players guard opens as soon as all the players are positioned, i.e. when they are all held on Wait_For_Restart.
At that point the referee is released and calls the procedure "Start", which sets the game state to "Play", opening this way the guard that release the players.
We also developed two procedures, Increase_Players_Number and Decrease_Players_Number, that actually play a smart role, because they help dealing with some complex situation with ease.
The clever idea behind these procedures is that if some player gets injured or sent off and leaves the field, he can just decrease the number of (active) players the referee will not wait for him anymore. Moreover, if a player gets injured or sent off, the referee will keep waiting for him until he leaves the field, because only at that point he will decrease the number of active players and the guard Wait_For_Restart'Count = Players_Number will automatically open. Because every player who leaves the game decreases that number, the semantic of Wait_The_End is clear: it just holds enqueued tasks until all the players have left.
The "Enable_Start_2nd_Time_Request" aims to solve an issue that comes out with a possible action of the simulator manager, since he interferes with the referee. The simulator manager can request to start the match in advance during the half-time break. If this request comes before some player arrives out of the field (and then are not yet held on Wait_For_Positioning), players would change movement and reach their kick-off position (on the other side of the field, clearly). But this would be a bit strange (the interval has been stopped before starting) and would also lead to some issue on the management of the half-time break timer, which is supposed to start only when all the players are out of the field. But the timer is managed by the referee, who can not wait until the players leave to start the half-time break timer and accept anticipated start requests at the same time. We found that delaying the request of an anticipated restart was better than allowing mixed conditions.

7.6.2 Detailed description of interactions and synchronizations

In this section we present a more detailed description of interactions between tasks and protected objects.
Sequence and interaction overview diagrams will be sometimes simplified to

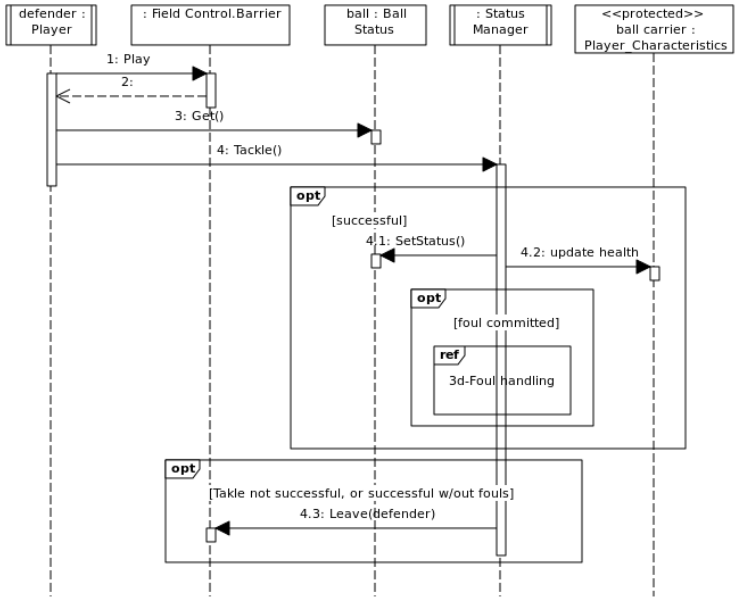

Figure 7.16: A player attempts a tackle.

focus on some aspects of the interactions. We consider two major cases. A tackle attempt, in which the player possibly causes a severe offence to the ball carrier, and a kick that sends the ball out of bounds, causing the referee to stop the game after being notified by his assistant. These two cases should be enough to explain how synchronization and inter-task communication works in all the other situations.
Figure 7.16 shows the first case. Note that the player's neighbourhood is released only after the ball status is updated. If the player committed a foul, then his neighbourhood is not even released by the status manager.

As shown in Figure 7.17, the call on "Leave" will be issued only after the referee finished managing the foul. Moreover (as shown in the previous figure), players read the game state after having granted access to the field and - as we explained - they cannot get access to the field until players in their neighbourhoods leave. This means that they cannot interfere with the player who made the foul, and when they start executing they see exactly what happened along with the updated game state. Consecutive fouls and other complex situations cannot happen (otherwise they would hang the system if the referee was already suspended on Wait_For_Players).

After having signalled the foul - or any other interruption (except the end of periods) - the referee waits until players reach their positions.
Players who were already executing (not nearby, clearly) ends their actions because they are not yet aware of what happened (and they will not interfere, anyway).

The interaction overview diagram in Figure 7.18 shows what happens afterwards. Timing is not expressed in the figure, so we would like to stress some

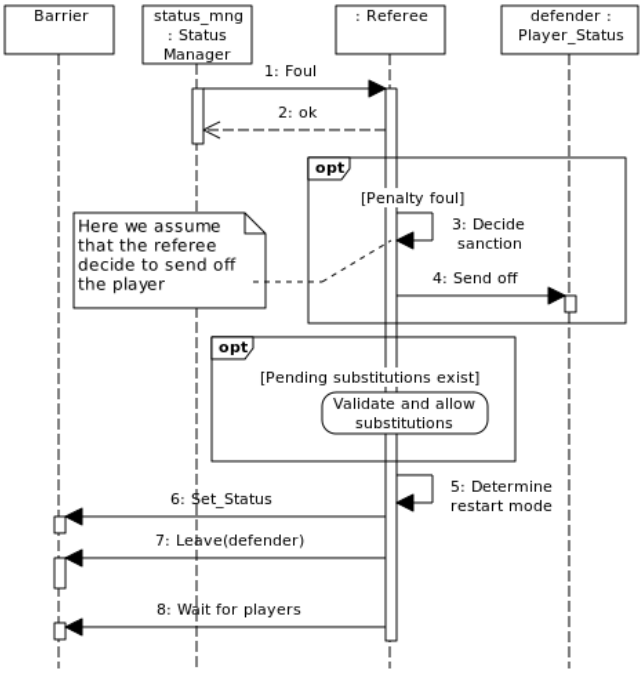

Figure 7.17: The referee handle a severe foul

aspects that might not be immediately clear. Players start executing with the rules explained so far. They will probably not execute during the same periods and they will generally execute their main loop many times.
Each time they start issuing a call on entry "Play", then they check the game state and their own condition. If something changed after the previous execution, they recalculate their movement and wait the next quantum. After the referee's signal the game state will be obviously changed, and also the personal condition of players who have been involved in the action (these updates are done before the referee's signal). During the next executions, the game and players states are not supposed to change. The players look around themselves to see if they have obstacles (other players) and eventually adapt their movement to adapt them, possibly (otherwise they wait for a while).
If they found a "convenient" position (i.e. the one they have chosen or one that at least brings them closer to their destination), they move forward and sleep for the time they will need to make the forthcoming step. If they have no other steps to do (because they reached their position or they are one step away but the one they wanted is occupied by someone else***), then they execute one of the interactions in the bottom-right corner of the figure. We use the "fork" symbol to indicate that those interactions are executed concurrently.
All of them refer to players in a certain condition.

The injured player (if any) will wait for a substitution until the end of the game. The diagram shows that afterwards the player loops back to "Check the game state". Then if he will find that the game state evaluates to "end match" (or "match aborted"), he will proceed with the termination procedure. Other-

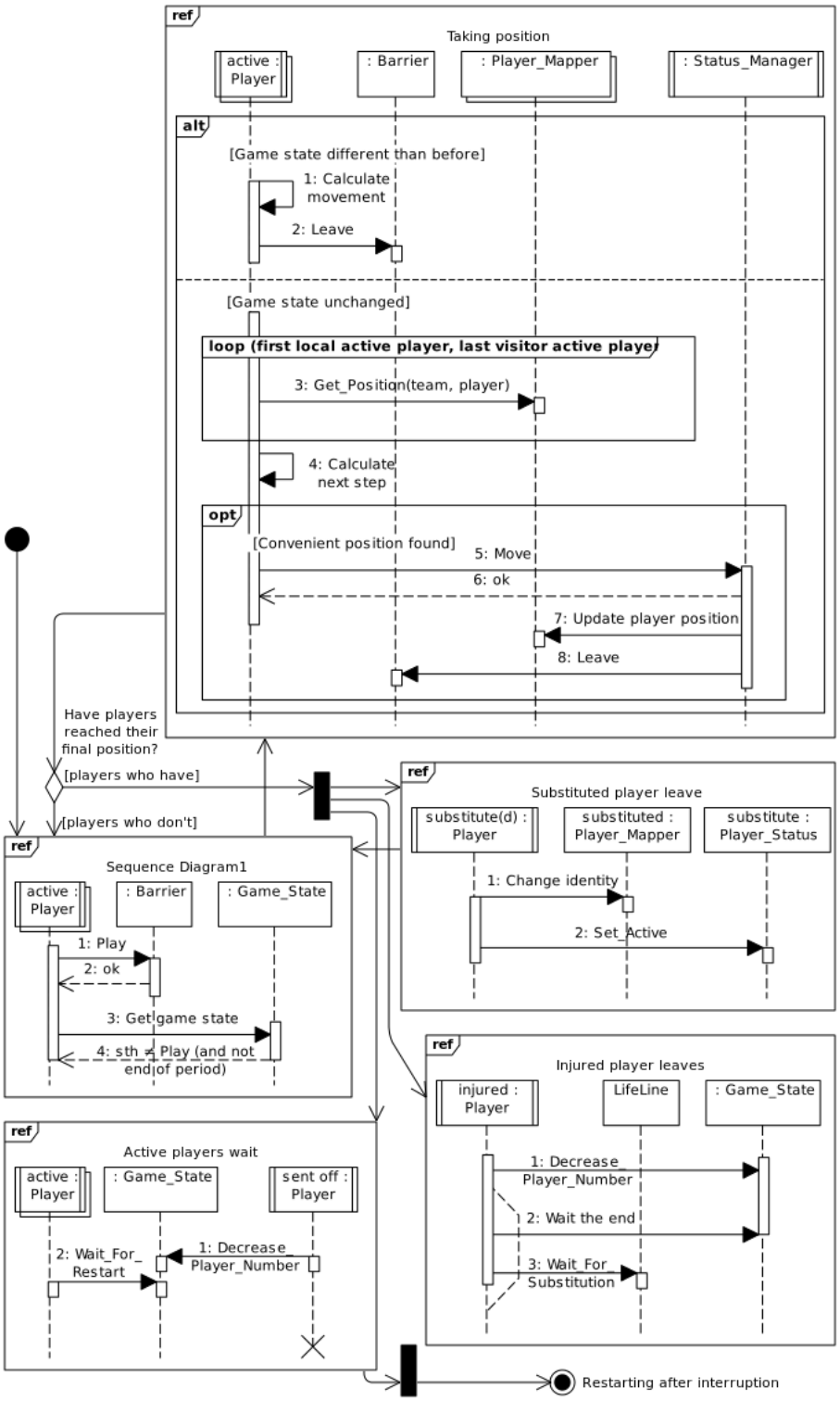

Figure 7.18: Players react to an interruption.

wise (if substituted), finding himself in the right position (out of the field), he will execute the "Substituted player leave" interaction as explained afterwards.

```
case Player_State is
   when Injured =>
      -- Prevent other players to pass him the ball
      Status.Teams_Status(Tid).Set_Role_Status(Pid, False);
      Game_Control.Game_State.Decrease_Players_Number;
      select
         -- Stop waiting in case of substitution
         Status.Players_Status(Tid, My_Id.Player_Number)
            .Wait_For_Substitution;
         Teams_Status(Tid).Set_Role_Status(Pid, True);
         Game_Control.Game_State.Increase_Players_Number;
      then abort
         -- wait until the game is over...
         Game_Control.Game_State.Wait_The_End;
         Match_Finished := True;
      end select;
```

The substituted player (there might actually be zero or more under substitution) change identity (taking the one of the substitute) and sets the status of the new identity (which was "inactive") to "active". The reuse of the same task is to avoid instantiating more tasks than those really needed. Afterwards he loops back to "Check game state". Finally, he will realise that his condition changed and recalculate his destination, consequently going back into the field before executing "Active wait sent-off leaves" (on the "active" players side).

```
   when Substitution =>
      -- update the global status of the player
      Status.Player_Mapper(Tid,Pid).Set_Numbers
        (My_Id.Player_Number,
         Status.Players_Characteristics
            (Tid,My_Id.Player_Number).Shirt_Number);
      Status.Players_Status(Tid, My_Id.Player_Number)
         .Set_State(Active);
      -- A player who enters in a role that was not covered because
      -- the substituted player was injured should notify that his
      -- role is now active again
      Status.Teams_Status(Tid).Set_Role_Status(Pid, True);
```

A player who has eventually been sent off decrease the number of active players and terminates. Active players instead (note, as wrote before, that they can become "active" after being "substituted") issue a call on "Wait_For_Restart". The sequence is analogue to one described in Section 7.6.1 and is shown in Figure 7.18. At the end, the player who is supposed to start passes the ball to a teammate. Handling interruptions is more complicated from players and referees' sides, because we also have to consider when special conditions hold (e.g. interruptions due to end of periods or penalty kicks in which players instead of passing the ball shoot it into the net). The complexity is mostly logical, so we will not go into deeper details.

Figure 7.20 shows the case in which a player kicks the ball out of bounds. The diagram should be enough self-explicative assuming what we already explained so far. The continuation "Referee interrupts the game" is similar to foul handling, but instead of (possible) assign a caution/sending-off and a free or penalty kick, the referee will assign a different appropriate "sanction". In case of throw-in, goal kick or kick-off, if the period time-out is over, the referee

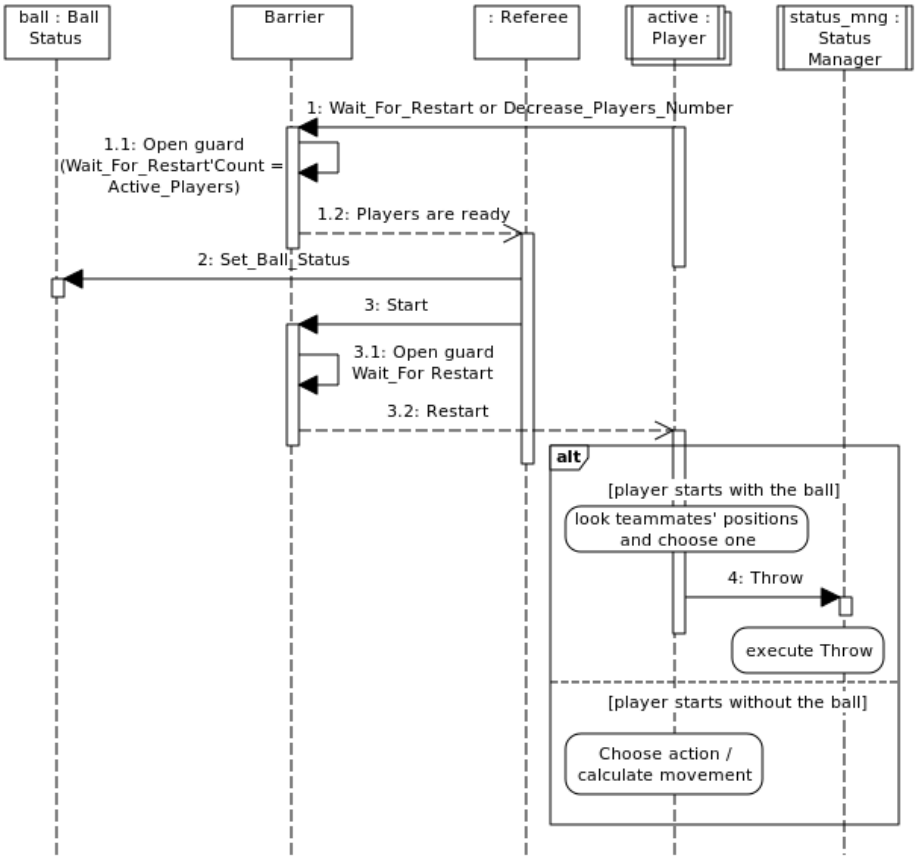

Figure 7.19: The game restarts after an interruption.

will stop the game.

7.7 Connecting the core with the world

Core subsystem has to distribute several information to GUIs to allow spectators and managers to watch a match properly. The most important information necessary to display a game in progress are the positions of the active players and the position of the ball on the field; this informaton is used by SpectatorGUIs to render the match, showing the game field with players and ball moving inside. The easiest way to do this is to periodically send a snapshot of the game field using the information maintained in the game status; receinving this periodic information SpectatorGUIs can easily deal with the game viewing. We decided to include in these snapshots the time information necessary to SpectatorGUIs to display the passage of game time. In the next section we introduce a task called Frame Manager which has the task of managing these snapshots (called television frames).
Other information to be distribute are players and team statistics; this information is used by ManagerGUIs and SpectatorGUIs to display the match statistics. Core subsystem has to keep this information updated based on the various events that happen during the game: players statistics are based on players actions (shots, passings, fouls, and so on..), teams statistics instead are calculated as sums of players statistics. As for television frames, our solution is to periodically send a statistics frame to distribution clients.
Finally, core subsystem has to distribute text messages to allow GUIs (particularly SpectatorGUIs) to print the commentary of the match. Here messages must be created and sent as soon as match events occur during the game.
Note that statistics update and messages creation occur as a result of specific events in the simulation, so we decided to opt for an event-based solution. In the next section we describe the software architecture implemented to accomplish this task.

7.7.1 The Frame Manager

As mentioned above, core subsystem has to periodically send a snapshot of the game field, containing the active players position, the ball position and the time information. To implement a single snapshot we use an Ada record type called Television Frame; in our solution we have a periodic task, called Frame Manager, which has the main purpose of send these frames to distributed clients.
We chose to keep a "current frame" within this task and to update it when players move instead of create a new frame from scratch every time. We decided to do so because we want that a frame represent a real snapshot of the game field containing consistent information inside. This is not possible if we create a frame every time reading the players positions from the game status because players can change their positions concurrently during the frame construction. The only way would be to have a read lock on all players positions during this operation but this is not desirable.
Hence in our solution Frame Manager task keeps the "current frame" updated with players movements notified by Status Manager task. As we will see with

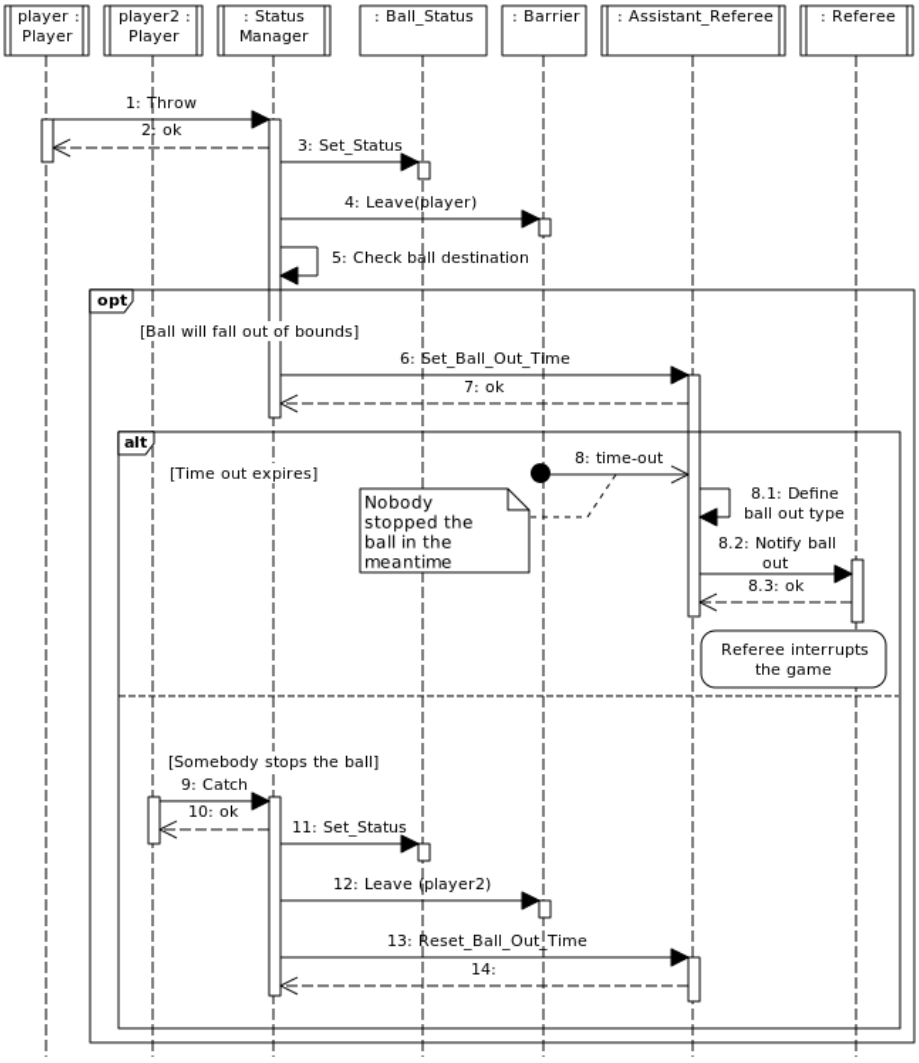

Figure 7.20: A player throws the ball, the linesman checks if it crosses the bounds to notify the referee.

this solution we are guaranteed that each frame sent has consistent information inside. Note that the ball position can always be read in a consistent way reading the game status because its position can change only at the beginning (or at the end) of a quantum and this guarantee that it is always consistent within a frame.
As already said Frame Manager has to send also time information inside every frame to allow SpectatorGUIs to display the game time. This information is the amount of time (in terms of minutes and seconds) passed from the begin of the current period (which can be first-half, second-half or break); Frame Manager computes this information and add it to the frame before send it to distributed clients. To do this properly Referee task has to notify the Frame Manager to say when current period starts/ends.

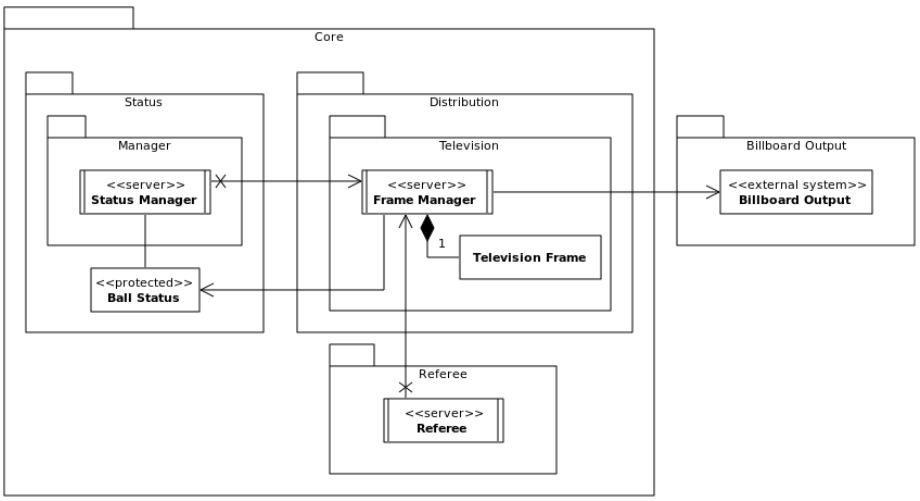

Figure 7.21: Architecture of the Frame Manager.

As we can see in the diagram, Frame Manager task exposes interfaces to Status Manager task - to update the "current frame" - and to Referee task - to compute the time information. Indeed Frame Manager exposes another interface used by Status Manager to notify the substitution of a player; this information is needed because every frame has the players shirt number inside. Finally Frame Manager exposes two other interfaces used to handle properly the initialization and the finalization of this task; to have more information about initialization and finalization of our system see Section 7.8.
We can sum up the main Frame Manager logic in cyclically waiting for:

- a notification from Referee to compute properly the time information, or
- a notification from Status Manager to update the players information, or
- the expiration of the task period (i.e. the frequency with which we send frames) to prepare and send the frame to distributed clients.

Here we can see a (simplified) portion of the Frame Manager task code:

```
-- wait for initialization signal
accept Start do
```

```
  -- init the "current frame"
end Start;
-- start the Frame Manager main loop
loop
  select
    -- wait for a request to restart the game timer
    accept Restart_Timer;
    [...]
  or
    -- wait for a request to stop the game timer
    accept Stop_Timer;
    [...]
  or
    -- wait for a player move
    accept Move_Player(Player_Team_Id : in Team_Id_T;
                       Player_Task_Id : in Player_Index_T;
                       New_Position   : in Coordinates_T) do
      -- update the player position in the current frame
      Television_Frame.Players_Array(Player_Team_Id,
          Player_Task_Id)
        .Player_Position := New_Position;
    end Move_Player;
  or
    -- or a player substitution
    accept Substitute_Player(Player_Identifier : in
        Identifier_T;
                             Player_Task_Id    : in
                                 Player_Index_T) do
      -- update the player shirt number
      Television_Frame.Players_Array(Player_Identifier.Team_Id,
          Player_Task_Id).Shirt_Number :=
        Status.Player_Mapper(Player_Identifier.Team_Id,
            Player_Task_Id).Get_Shirt_Number;
    end Substitute_Player;
  or
    -- wait for the finalization signal
    accept Quit;
    exit;
  or
    -- wait for the expiration of the task period
    delay until Next_Television_Quantum;
    -- write the time information into the current frame

    -- add the ball position to the current frame
    Television_Frame.Ball_Position :=
      Motion.Get_Current_Position(Status.Ball_Status.Get.Motion
          );

    -- send current frame to billoard output
    Send_Television_Frame(Television_Frame);

    -- set the new Next_Television_Quantum
    Next_Television_Quantum :=
      Next_Television_Quantum + Television_Frequency;
  end select;
end loop;
```

As we can see when the task period ends Frame Manager prepares the frame (adding the time information and the ball position) and send it to the distributed clients. This operation occurs during the execution of the delay alternative and so, because the other entries are not available, the current frame can't be modified during the execution of this code. This guarantees that the information

inside each frame sent are always consistent and so correspond to a real snapshot of the game field.

7.7.2 The Event Manager

As already said statistics update and messages creation occur as a result of specific events in the simulation. For example, when a foul occurs core subsystem has to create and send the massage to allow GUIs to "say" that a foul has occurred and to update the players and teams statistics (in this case the number of fouls done by the offending team and the number of fouls suffered by the defending team). Hence we decided to opt for an event-based solution: when an event occurs in the simulation, core subsystem processes this event doing what the event requires (sending the foul message and updating the statistics in the previous example).
We decided to keep separate the events creation and the processing of these events: in our solution we have many events producers and one event consumer (processor) "talking" to each other using an events buffer, thus realizing the well known producer-consumer pattern. The event buffer is realized as a protected FIFO queue to guarantee that events are processed in the correct order (i.e. in the order they are created). Events producers are all those active entities which cause events in the simulation (i.e. Status Manager, Referee, Assistant Referee, Players, etc); events consumer is a task called Event Manager which has the task of process events when they occur. Note that with this solution we assign to the Event Manager the task of keeping statistics up to date; because it contains all the data structures needed, we decided to use Event Manager task also to periodically send the statistics to distributed clients (hence Event Manager is a periodic task).

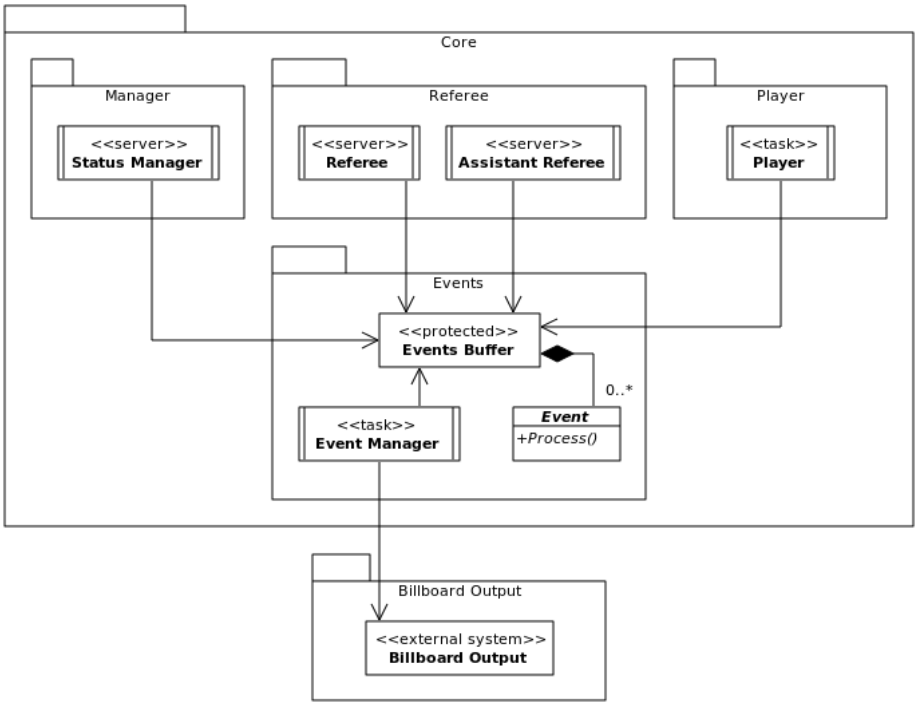

Figure 7.22: Architecture of the Event Manager.

When Event Manager gets a new event from the buffer he has to process it, doing operations depending on the type of the processed event. These operations can be the updating of the statistics and/or the sending of a message. To implement this solution properly we realized a hierarchy of events; the root of this hierarchy is an abstract event with an abstract procedure to process the event. Each child event implements this procedure to realize the processing of the particular event and add to the event data structure any other data needed. Here we can see a class diagram of the events hierarchy, showing the most important events created during the simulation.

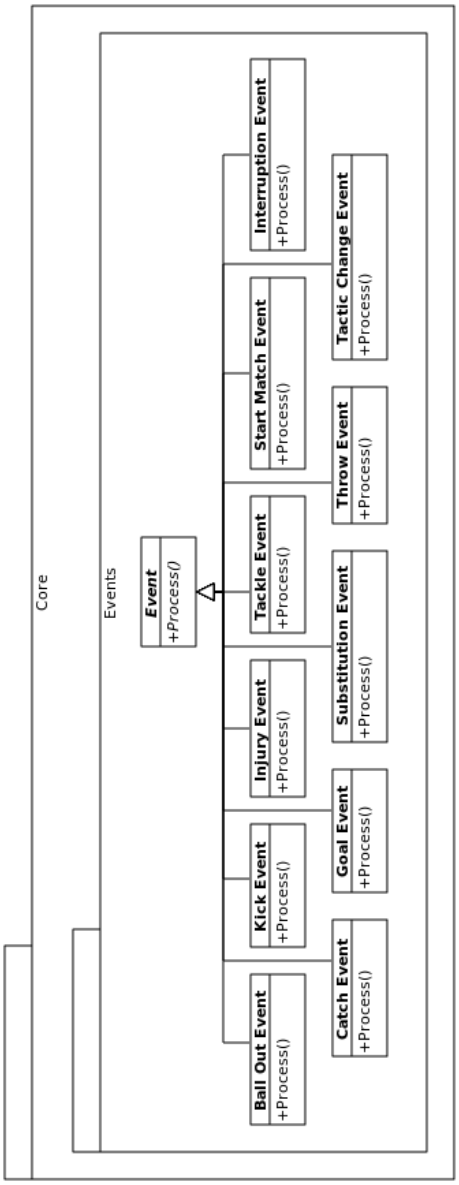

Figure 7.23: "Event" type hierarchy.

As previously said Event Manager has two tasks: to process events when they occur (i.e. when he finds events in the buffer) and to periodically send statistics to distributed clients.
We can sum up the main Event Manager logic in cyclically waiting for:

- a new event in the buffer to be processed or
- the expiration of the task period (i.e. the frequency with which we send statistics) to send the statistics.

Event Manager task exposes only one entry to start his main loop; here we can

see a (simplified) portion of his code:

```
-- wait the start signal
accept Start;
while not Has_To_Terminate loop
  select
    -- wait for an event in the buffer
    Events_Buffer.Wait_For_Event;
    -- here we have at least one event in the buffer, take the
        first
    -- and process it
    Process(Events_Buffer.Get_First);
    -- after processing, delete processed event from buffer
    Events_Buffer.Delete_First;
  or delay Statistics_Period;
    -- send all statistics to "distribution"
    Send_Statistics_Frame(Statistics);
  end select;
end loop;
```

As we can see Event Buffer exposes an entry *Wait_For_Event* used by Event Manager to wait until an event arrives; this entry has a barrier which is open when the event buffer is not empty and closed otherwise.
Messages are sent by Event Manager as simple strings; statistics sending is obviously more complicated. Statistics are implemented as Ada record type; particularly we use a record containing all match statistics, those of players and teams. As already said Event Manager task keeps updated this structure processing simulation events and periodically send it to billboard subsystem. Note that we do not need to protect this data structure because the only entity which read and write it is the Event Manager task, which is a singleton task and so can not work concurrently.

7.8 Initialization and finalization of core subsystem

Core subsystem has several tasks to do before a match can begin and to properly terminate it. To properly start a simulation core subsystem has to load the simulation and teams configurations. As we already said we want that core waits for a possible SimulatorGUI configuration, without being blocked if no simulator user connects to the system. Moreover we want that core uses the configuration sent by ManagerGUIs for teams initialization, using the default configuration if no ManagerGUIs connects to the system.
As regards system finalization it is not possible to simply "let the system terminate"; we will see that we have to coordinate the termination of active entities of the core subsystem.
In the next sections we expose the initialization and finalization procedures used in our solution.

7.8.1 Core initialization

We can sum up the initialization phase in these steps, implemented by the main procedure:

1. creation of the core entities: all entities are created in the declarative part of the main procedure. After the begin, Players tasks wait for the start signal by issuing a call on the entry *Barrier.Wait_For_Positioning*; reactive entities wait for the initialization on their accepts;

2. loading of the default configuration: core subsystem needs two configuration files used to store the default configuration (simulation.xml for simulation configuration and teams.xml for teams configuration); in this step core loads the configuration from these files;

3. waiting for GUIs configuration: after loading the default configuration core has to wait for the SimulatorGUI configuration; if no configuration arrives before the timeout expiration core uses the default configuration previously loaded. The same occurs with teams configurazion, possibly sent by ManagerGUIs;

4. start of the match: finally core can send the start signal to the other entities waiting for the initialization, particularly the Referee which can now start the match.

From the point of view of concurrency design step 3 is the most interesting because it includes the synchronization of various tasks. As already said core has to wait for SimulatorGUI configuration, without being blocked if no simulator user connects to the system; moreover we want that after the time-out expires core refuses any configuration sent by SimulatorGUI. We realize this using a protected object called Simulation Configuration, containing data structures and procedures/entries used for the synchronization. The most important are:

- entry Wait_For_Simulation_Start: this entry is used by main procedure to wait for the start signal sent by SimulatorGUI (along with the simulation configuration). The entry has a barrier which is closed until no SimulatorGUI arrives; it is used in a select statement with a delay alternative to implement the time-out;

- procedure Set(...): this procedure is used by "distribution" to set the simulation configuration (when it arrives from SimulatorGUI); the operation succeed only if core accepts the configuration (i.e. the time-out is not already expired);

- procedure Stop_Receiving: this procedure is used to stop the receiving of the configuration (as a result of the time-out expiration); it is called by main procedure after the select statement.

We used a similar solution for teams configuration; here the difference is that core has not to wait for ManagerGUIs: if configuration arrives before the time-out core uses it as team configuration, otherwise it uses the default configuration (previously loaded in step 2). To implement this solution we used an other protected object used to guarantee data consistency, called Teams Configuration; in similar manner to Simulation Configuration, also this protected object contains a procedure Set(...) used by distribution to set teams configuration (if time-out is not expired) and a procedure Stop_Receiving used to stop the receiving of configuration after the time-out expiration.

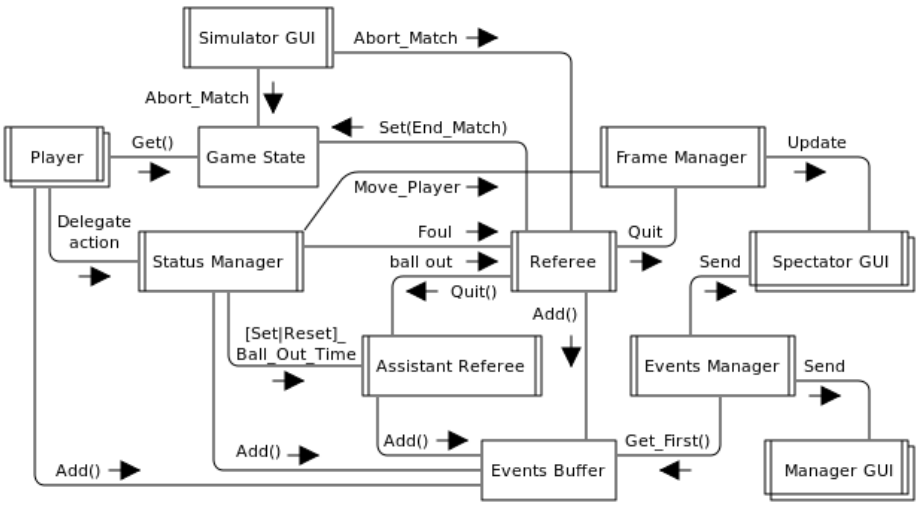

Figure 7.24: Inter-task communication and access to protected objects.

7.8.2 Core finalization

Two events cause the finalization of the core subsystem: the end of the match - in which the finalization is delayed after all the players have left the field - and the abort request sent by the Simulator user. The abort also causes a soft termination, but its execution is almost immediate.
Generally, the chosen method of termination mostly depends on the characteristics of each task:

- Players are active tasks, so they should terminate by themselves; in our design there are special game states for the end and the abortion of a match;
- Status Manager is a reactive task, provided of an "or terminate" alternative so that it can terminate automatically when all the other tasks (that can possibly call it) are terminated (or on a "terminate" alternative);
- Assistant Referee, Referee and Frame Manager are reactive task with "or delay" alternatives, so the "or terminate" choice was not available. We notify them the termination via rendez-vous; note that if the match finishes regularly (i.e. without "abort" requests), the referee manages termination by himself;
- the events (and statistics) manager is an active task; we created a special termination event that - when processed - causes the task termination.

The most important aspect to care about in finalization design is probably the need to avoid circular wait. Figure 7.24 shows the direction of inter-task communications. In case of regular end of the match, the sequence is as follows:

1. The referee waits until the ball goes out of bounds, then realizes that time is over and signals the end of the match.

2. During their next main cycle, players detect the new game state and move outside the field. Then each player decreases the number of active players and terminate.

3. In the meantime, the referee waits until all the players have left through the "Game_State.Wait_The_End" entry, which opens when the number of active players becomes zero. Having the referee waiting on "Game_State. Wait_For_Players" would be the same, because when all the players have left, the number of active players becomes equal to the number of player tasks enqueued on "Wait_For_Restart" (zero), thus opening the guard.
   We need the "Wait_The_End" entry also for injured player, because they wait until the end of the match if they do not get substituted before; queuing them on "Wait_For_Players" would result in awaking them at every interruption and would make implementation less clear; this mechanism works for injured players because they decrease the number of active players as soon as they leave the field (re-incrementing it if they get substituted).

4. Afterwards, the referee notifies his assistant and the Frame Manager by invoking their "Quit" entry. Before allowing the referee to rendez-vous his assistant (thus avoiding possible deadlocks) we must ensure that the latter does not invoke any entry on the referee. This condition is guaranteed because the ball is already out of bounds (and the assistant referee already notified the referee about the occurring condition, causing him to end the match). Nothing else could cause the assistant referee to call the referee, so the communication is safe.

5. Finally, the referee add the special termination event to the Event Buffer causing the termination of its consumer task. This mechanism is also safe because the other tasks have already been terminated, so they will not generate any further event (this would raise a run-time error).

Abortion is similar, but it is initiated by Simulator User, whose interface calls the method "Stop_Match". The method's body change the Game State to "Aborted". This causes players to terminate at their next main loop (when they detect the occurring condition), but this time without leaving the field. "Stop_Match" waits until they leave the game decreasing the number of active players (in the previous case this was done by the referee).
At that point the method call the Assistant Referee task's "Reset_Ball_Out_Time" entry, which invocation causes it to close its guard on the "or delay" alternative (i.e. the assistant referee stops watching the ball). Since all the players tasks are terminated, they cannot send the ball out (thus delegating the Status Manager to re-activate the Assistant Referee).
Finally, the remote method invokes the entry "Quit" on the Referee, which causes him to proceed with termination, initially waiting until players leave the game (but the guard is already open), then proceeding with the steps 4-5 above.

Chapter 8

Design of Billboard subsystem

8.1 Core Partition

On a distribution approach core contains a Remote Call Interface **CoreListener**, which defines all the remote procedures. This procedures are remotely called by BillboardInput to notify GUI requests.

These are all the possible different requests:

1. GetInitialConfiguration
2. StartMatch
3. StopMatch
4. StartSecondHalf
5. GetManagerConfiguration
6. SaveTeamConfiguration
7. TacticChange
8. PlayerSubstitution

In response to this requests, the core calls NotifyResponseInquiry defined in RCI BillboardOutput (located in BillboardOutput partition).
We used the procedure NotifyResponseInquiry as a facade to achieve decoupling.

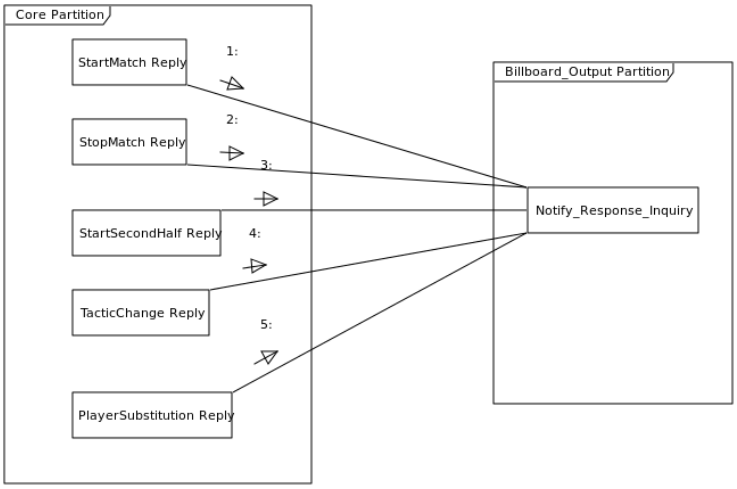

Figure 8.1: Distribution, response Inquiry facade

8.2 BillboardOutput Partition

BillboardOutput contains a Remote Call Interface **BillboardOutput** that defines all the remote procedures. These procedures are remotely called by Core and BillboardInput partitions.
When BillboardOutput receives Remote Procedure calls, it transforms its Ada data types into YAMI4 types. Then it publishes an event on the indicated channel.

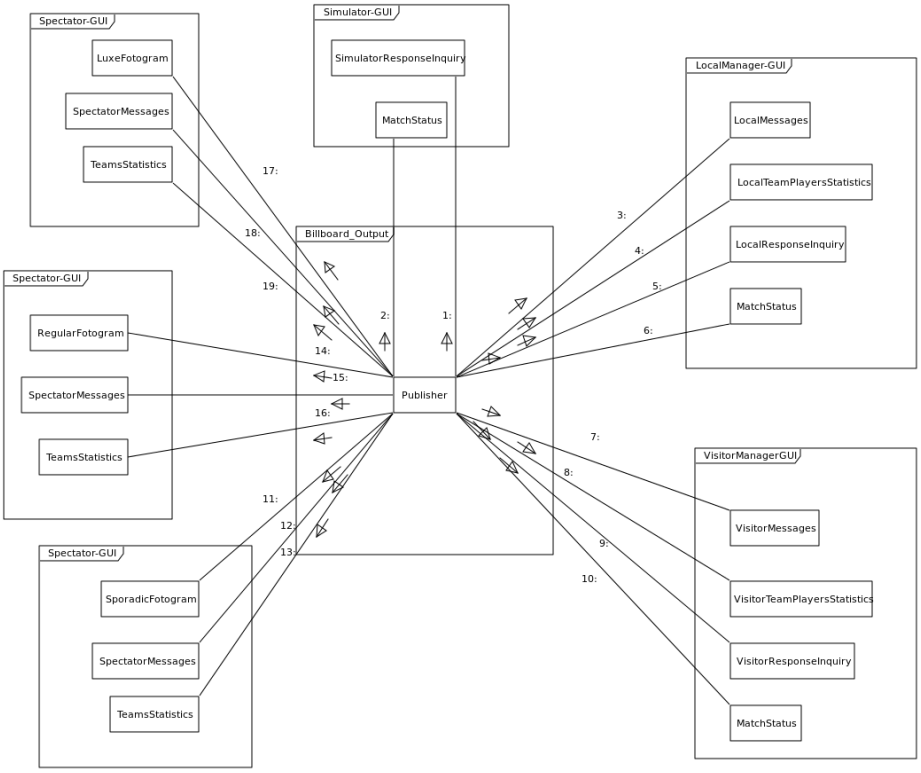

Figure 8.2: Distribution, BillboardOutput

BillboardInput partition calls the remote NotifyResponseInquiry procedure.

These procedures are called by Core partition to notify Core events:

1. SendTelevisionFrame
2. SendStatisticsFrame
3. SendMessage
4. NotifyResponseInquiry
5. ResponseManagerSave
6. NotifyMatchStatus
7. NotifyStartedLineup
8. NotifyInitialConfiguration

These procedures are used by BillboardOutput business logic:

1. NotifyMQTelevisionFrame
2. NotifyLQTelevisionFrame
3. InitializePublishers

InitializePublishers creates all the resources needed for Output communication.
BillboardOutput partition has two periodic tasks; these tasks send out television snaphots at lower frequences than the predefined one used by the Television Manager described in the previous chapter.

8.3 BillboardInput Partition

This partition receives all GUI requests.
If it receives a connection request, it replies immediately to the caller (synchronously).

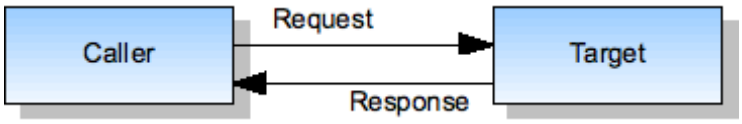

Figure 8.3: Distribution Client Server [13]

For asynchronous requests types, it transforms all YAMI4 data parameters into our Ada data types.
Then it calls the requested remote call procedure. This partition can call procedures contained on Core or BillboardOutput partitions.

These are all the different requests:

1. connect
2. disconnect
3. startMatch
4. stopMatch
5. startSecondHalf
6. saveTeamConfig
7. tacticChange
8. playerSubstitution

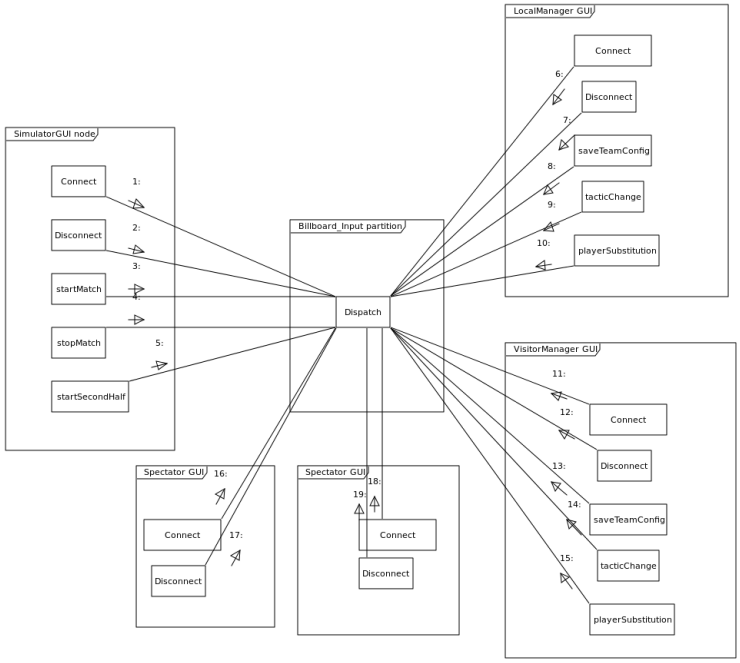

Figure 8.4: Distribution, BillboardInput

8.4 Java GUIs:

The first one is **SimulatorGUI**, which receives notifications, after subscription, using the Publish-subscriber paradigm. SimulatorGUI makes requests following the Client/Server paradigm. YAMI4 libraries already implement both mechanisms.
SimulatorGUI is subscribed to BillboardOutput publishers.
These are the communication channels:

MatchStatus: through this channel, GUIs are notified about all match status changes like WarmUp, MatchStarted, SecondHalfStarted, FirstHalfFinished, SecondHalfFinished and MatchAborted;

SimulatorResponseInquiry: through this channel, the core is notified about all the previous SimulatorGUI requests and also receives notifications about the number of the current connected nodes;

SimulatorGUI also performs requests to BillboardInput partition using YAMI4 client-server mechanisms. So the Client part of SimulatorGUI sends its requests to the Server part of the BillboardInput partition. The possible requests are:

1. **Connect:** This request is the unique synchronous call, implemented by the OutgoingMessage method. Here BillboardInput validates the connection, just accepting the first "SimulatorGUI" client.
   If the request is valid BillboardInput replies with the YAMI4 state OutgoingMessage.MessageState.REPLIED.
   If the request is not valid BillboardInput rejects the connection with the YAMI4 state OutgoingMessage.MessageState.REJECTED.

2. **Disconnect:** This is an asynchronous request; it notifies that SimulatorGUI is going to close.

3. **StartMatch:** This asynchronous request notifies that the user wants to start the Match. The response will arrive on the SimulatorResponseInquery channel.

4. **StopMatch:** This asynchronous request notifies that the user wants to stop the match. The response will arrive on the SimulatorResponseInquery channel.

5. **StartSecondHalf:** This asynchronous request notifies that the user wants to stop the match. The response will arrive on the SimulatorResponseInquery channel.

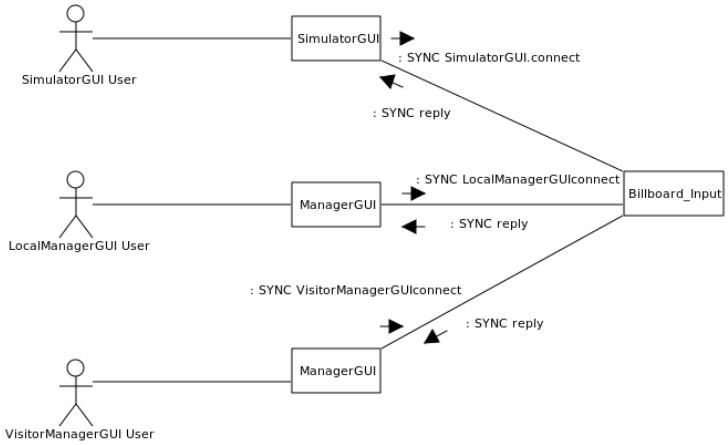

Figure 8.5: Distribution, GUI synchronous communication

The second and third nodes LocalManagerGUI and VisitorManagerGUI also receive notifications using the Publish-subscriber mechanism, and make requests using the Client-Server mechanism. We use the YAMI4 libraries that already implement both mechanisms.

LocalManagerGUI is subscribed to BillboardOutput Publishers; these are the communication channels:

MatchStatus: This is the same channel mentioned above in simulatorGUI.

LocalTeamPlayersStatistics: This channel is used to notify Statistics related to the local team.

LocalMessages: This channel is used to send text messages that will be displayed on the GUI.

LocalResponseInquiry: on this channel is notified about all previous LocalManagerGUI requests.

LocalManagerGUI also performs requests to BillboardInput partition using YAMI4 client-Server mechanism. So the Client part of LocalManagerGUI sends its requests to the Server part of the BillboardInput partition. This is the requests list:

1. **Connect:** This request is the unique synchronous call, implemented with OutgoingMessage method. Here BillboardInput validates the connection just accepting the first "LocalManagerGUI-Client". If the request is valid, BillboardInput replies with the yami4 state OutgoingMessage.MessageState. REPLIED. If the request is not valid, BillboardInput rejects the connection with the yami4 state OutgoingMessage.MessageState.REJECTED.

2. **Disconnect:** This is an asynchronous request; it notifies that LocalManagerGUI is going to close.

3. **SaveTeamConfig:** this is an asynchronous request; it notifies that the user wants to save the team configuration. The response will arrive on the LocalResponseInquery channel.

4. **TacticChange:** This is an asynchronous request; it notifies that the user wants to make a Tactic Change. The response will arrive on the LocalResponseInquery channel.

5. **PlayerSubstitution:** This is asynchronous request, which notifies that the user wants to make a Player Substitution. The response will arrive on the LocalResponseInquery channel.

VisitorManagerGUI is subscribed to BillboardOutput Publishers, these are the communication channels:

MatchStatus: This is the same channel mentioned above in simulatorGUI.

VisitorTeamPlayersStatistics: This channel is used to notify statistics related to Visitor team.

VisitorMessages: This channel is used to send text messages that will be displayed on the GUI.

VisitorResponseInquiry: on this channel is notified about all previous VisitorManagerGUI requests.

VisitorManagerGUI also performs requests to BillboardInput partition using YAMI4 client-server mechanism. So the Client part of VisitorManagerGUI sends its requests to the server part of BillboardInput partition. This is the requests list:

1. **Connect:** This request is the unique synchronous call, implemented by the OutgoingMessage method. Here BillboardInput validates the connection, just accepting the first "VisitorManagerGUI" client. If the request is valid BillboardInput replies with the YAMI4 state OutgoingMessage.MessageState.REPLIED. If the request is not valid BillboardInput rejects the connection with the YAMI4 state OutgoingMessage.MessageState. REJECTED.

2. **Disconnect:** This is a asynchronous request; it notifies that VisitorManagerGUI is going to close.

3. **SaveTeamConfig:** this is asynchronous request; it notifies that the user wants to save the team configuration. The response will arrive on VisitorResponseInquery channel.

4. **TacticChange:** This is an asynchronous request; it notifies that the user wants to make a Tactic Change. The response will arrive on VisitorResponseInquery channel.

5. **PlayerSubstitution:** This is an asynchronous request; it notifies that the user wants to make a Player Substitution. The response will arrive on VisitorResponseInquery channel.

The fourth node, **SpectatorGUI**, receives notifications using Publish - subscriber mechanism.
SpectatorGUI is subscribed to BillboardOutput Publishers; these are the communication channels:

FotogramFrame: this channel is used to notify fotogram frames; each SpectatorGUI can choose one of three different possible frame rates. This feature aims at dealing with different network speeds.

TeamsStatistics: this channel is used to notify game statistics.

SpectatorMessages: this channel is used to notify game messages, practically the commentary of the game.

8.5 Distribution Summary (historical narrative)

- Analysis and Design

  At the beginning we choose to use one node to Core and other node to Billboard and one node for each front-end client.
  Our first choice was to use CORBA to communicate the entire simulation system. The reason was that our system is not heterogeneous, because we had already decided to use Java language to build the front end.
  Then we chose to adopt the PUSH model to output from the core and also to send messages to the core.

  Then we understood that we can use Ada in Core node and also on the Billboard node, so we understood that it would be simpler to use DSA and create a homogeneous Core-Billboard system.

- Implementation

  IDL Then we began to write IDL data structures, that will be used like Value Objects between Billboard and front-ends. We compile these IDL files in Java and Ada, then we saw that code generated by IDL was really similar to the objects that Ada should use in its remote procedures calls.
  So we try to use the same "Ada data structures" generated by IAC compiler, like value-objects parameters in the procedures declared in our defined RCI package.
  We look that IAC generates data structures based on CORBA types and that was not the same types that a regular DSA use.
  We try to define some remote types using CORBA types but we had no success. We try to search through the Internet some help to solve these problems, and we found just a few comments about this issue, so we decided to abandon it momentarily.

Polyorb We downloaded Polyorb and installed it like other programs just using "make install" and the installation ended without errors.
Then we run the Polyorb examples and we understood that it did not run correctly.
We read many documentation about the Gnatdist macrocompiler, trying to understand why it did not work; one more time we searched on the Internet, with no success.
Then we wrote on the **ada IRC channel** for some help and they gave us the correct installation parameters.
We coded our adaway.cfg and made some tests with basic integers and string parameters.

Java We read some Java documentation and examples, then we noticed that Java (orbd included in the JDK distribution) just implements a small part of the CORBA specification, so the Event Service and Notification Service are not implemented. We also saw that all samples were old.
We looked for the other CORBA implementation providers; we found that nobody gives continuity to the old Open Source projects like Openorb. In the business market Oracle practically bought all the other vendors like BEA Tuxedo or Glassfish-CORBA. We understood that we were dealing with a market issue.
So we write on the **Polyorb mailing list** asking if anyone had been able to use the event service between Java and Polyorb before.
They answered with *"the biggest problem is that even though all vendors have good intentions and believe in the seamless interoperability between different products, whenever something bad happens on the line between two unrelated implementations, you will find it difficult to get all those vendors together to help you."*[16]
He suggested use YAMI4, we read that last year was presented on Ada-Europe, so we decided to use it.

Yami4 Then we installed YAMI4 and we tested it on Java; it worked just like any other API.
For Ada it was a little bit different because at first we had to solve a library linking problem.
Then we were no really sure how to use an external library; at first we tried to include it like a library but we got some linking problems.
Then we wrote to the **YAMI4 mailing list** and they suggested to include all YAMI4 sources in our project because otherwise we would get always the old linking problem. We did not agree but we did not know how to solve it in a different way.
Then we had some little issues like synchronization but they were solved quickly.

Integration Problems We had the same equivalent types defined in both the Core partition and Billboard partition; we combined and placed them inside the Billboard partition.
Then we saw that we had a circular dependency problem; when Core starts up, it needs to have Billboard already started and when Billboard starts up needs to have Core already started. So we decided to split Billboard in two partitions: BillboardInput and BillboardOut-

put.
When we integrated the whole Core within a distributed project we have many dependency problems; to solve them we needed to define a common directory that should be included by BillboardOutput, BillboardIntput and Core Ada projects. the fastest solution was to include all the directories in all projects.

Chapter 9

Design of the distributed Graphical User Interfaces

The soccer simulation system, as a distributed system, is based on a 3-tier client/server architecture: the server (Core) is the component that encloses the logic of the simulation, it is the main source of events. The Billboard intermediator deals with the communication between the Core and the Front-ends. Billboard, on GUI's perspective, is the unique interface used to know about the system. Clients (Front-ends) use the services offered by Billboard, they enable users to see, to control and to take part to the soccer simulation.
This chapter focuses on the design of the Graphical User Interfaces (GUIs from now on). The first part explains from a high level perspective how the requirements have been grouped to match logical-related features within the system. Motivations of the key design decisions are provided subsequently, together with the formulation of protocols needed to the cooperation between GUIs and Billboard. The second part describes the detailed design for each category of GUI. We discuss the internal multi-thread system, the packages structures along with the role of each class.

9.1 High level design

9.1.1 Categorization of requirements

The requirements for the soccer simulator system are many and diverse, in order to reduce the complexity following a classic top-down approach, they have been categorized into three groups of logical-related functionalities (requirements map to features offered to the user). The functionalities of the first group are related to the visualization of the simulation, so the players, the ball and the soccer match simulation statistics. The second group is about functionalities that aims to control the flow of simulation, start/abort signals, including the configuration of the simulation characteristics. The third group includes functionalities related to the management of the team who participate to the simulation. The user should be able to select the team, players and the tactic before the beginning of the simulation. During the match he/she can change the tactic or substitute a player, there is also a necessity to visualize the status/statistics of the players

as they play.
The previous categorization leads to the identification of the three correspondent category of users, named respectively: **Spectator User**, **Simulator User**, **Manager User**. We decided to design a separate GUI with separate features for each category of user. We name the interfaces **Spectator GUI**, **Simulator GUI** and **Manager GUI**. Since the system has to be distributed, we allow the users to instantiate the preferred combination of GUIs on a particolar node. For example, instead of replicating the code that implements the visualization of the match on the Manager GUI, the Manager User can simply open an instance of Simulator GUI and view the match from there.

9.1.2 General considerations

Implicit requirements must be analyzed once we decide to separate the GUIs in such a way. The maximum number of allowed Simulator GUI within the system have to be one because conceptually exists only one center of control for the simulation. The maximum number of allowed Manager GUI must be two, one for the local team and the other one for the visitor team. The Simulator GUI instances should not be bounded in multiplicity because their only concern is to show the match and the statistics.
Regarding the communication's side of the GUIs, we can model Simulator GUI and Manager GUI having a bidirectional channel. The input channel serves to acquire information from the system, the output channel is needed to send actions to the system, triggered by the user. Differently, the channel can be unidirectional (input only) for the Spectator GUI because it only needs to acquire data in order to display it in a convenient way.
Extra care must be taken because we don't want synchronous coupling due to the GUIs. What we want to achieve is an asynchronous robust interaction, the system must be autonomous, being able to start the simulation with a default configuration without relying on GUIs presence. On the other hand, we want GUIs that not block in a synchronous way waiting for the notifications from the system. A poor structuring of the flows of control lead most of the time to irresponsiveness. Transient freezing of the graphic occours beacuse that particular flow of control is blocked waiting for the response instead of being available to serve user's action.
In addition, we need to define a model of communication between GUIs and Billboard, that is the interface with the rest of the system. The Billboard interface is divided into two partition: the input partition (flow of data from the GUIs to the system) and the output partition (flow of data from the system to the GUIs).
GUIs issue requests to the Billboard input partition using an usual client/server mechanism, while the other direction can be modeled using publish/subscriber model. Billboard output will care about the publishing part while the GUIs will play the subscriber part, they register for events of interest and take subsequent actions triggered by that events. The useful properties are that the subscriber doesn't need to use polling, nor the publisher needs to have separate connections towards the GUIs. The subscriber registers a callback point to be called when the requested information are available, Billboard simply has one unique channel for that type of notification. We can reuse the channel subscribing to it from another GUI, reducing overall dependencies and yielding to a more flexible

system.
GUIs are basically state machines, we define the standard behaviour for each GUI defining its states and transitions. The formulation of the states can be tricky because often includes information that we don't have already, we have to acquire it from the Core. Especially the informations required for the initialization (no hardcoded settings), has to be loaded from the outside. Transition between states are caused by the user, or by incoming notifications.
We should introduce a protocol, to keep things easy, that enables us to standardize the behaviour of the GUIs. Protocols are really useful to define exactly how the exchange of the information with Billboard is handled. Here we discuss the general mechanism. The detailed design will be provided for each typology of GUI in the second part of this chapter.

9.1.3 GUI's initialization

We must consider the initialization of the GUIs first. This means the exchange of data needed to the GUI to be fully working. We divide into three sequential step:

1. activation
2. connection
3. initialization

The GUI is instanced by the user in the activation step, it builds its own internal structures and becomes ready to be connected with the rest of the system. In the connection step the GUI make aware of his presence calling Billboard input partition, validation constraints are evaluated here. In the third step the GUI is allowed to proceed, it receives the initialization from the system through the Billboard and becomes fully working within the system.
The activation step can be followed automatically (without user intervention) by the connection step on Spectator GUI and Simulator GUI. A different situation regards Manager GUI, beacuse we have two roles of the same interface: locals or visitors. So we let the user to choose the preferred role before connecting.
During the connection step, Billboard enforces the constraints on the maximum number of GUI instances, so according to these constraints a GUI can pass successfully the connection step (allowed), or not (rejected). Here, we are forced to use a synchronous mechanism to ensure that proper response will be routed to the right GUI. The publisher/subscriber model doesn't fit well for this task, so the reply will be received on the same thread as does the return of a usual call. We introduce a time-out in case of Billboard failure, so the thread of the GUI that made the call does not block indefinitely. Once the GUI has been rejected it remains isolated from the system, the only action that can be carried out is to close it.
If the GUI is allowed to proceed, the subscribers of that GUI register to the appropriate channels exposed by the Billboard output. After that, one of them will be notified with the initialization data from the system. The initialization data are represented by two xml files and by the state of the Core at the request moment. The first xml file holds the possible values for the configuration of the simulation (time duration values, break time values, etc.), the second xml

file holds the teams settings (teams, players, formations, pressing level, etc.). GUIs need this information for the population of their graphical components (combo boxes, tables, soccer field widget) and for triggering the appropriate initial state, since we allow the fact that the user can at anytime instance and close the GUIs.
However, exceptional situation must be resolved. If Billboard input partition is not available during the connection step we let the GUI retrying to connect after five seconds for an unlimited number of times, until the user decides to exit the GUI. If Billboard output partition or the Core are not available during the initialization step, then the GUI cannot initialize: the user is forced to start the missing components and to restart the GUI.

9.1.4 GUI's normal operations

Successful initialization brings the GUIs to their operative state. We identify the set of required operations for each GUI's category, then we describe how the requests (omitting connection on instantiation and disconnection on closure) and notifications are handled.
Spectator GUI doesn't output any request, it only receives informations about the match such as the snapshot of the players and ball, the match statistics and commentary. The Spectator GUI's subscribers receive live notification exclusively during the simulation. The user can see on the interface the players as they play.
Simulator GUI outputs the following requests: start match, start second half, abort match. It has to be able to receive the replies of such a signals and also the notifications of the state of the simulation. The client part of the GUI issues asynchronous requests based on user inputs, the replies are received by the response subscriber of the GUI. We design the GUI to reflect the current state of the simulation at any time: the match state subscriber receives the notification and triggers the appropriate state. The simulation states are: warm up, first half, break time, second half, end match. During warm up, the user can make the configuration (period duration, break duration, maximum allowed substitutions) and then start the match, if he/she waits too much causing the expiration of the Core initial countdown then the simulation begins automatically with the default configuration. The first half state is notified. During first half state, the user can abort the match. When the first half ends, the break time state is notified. During the break time the user can abort the match or force the start of the second half. If the break countdown succeeds then the second half is resumed automatically. The second half state is notified. During second half state the user can abort the match. When the second half ends, the end match state is notified. The end match state and the state following a successful abortion are the same. When the simulation ends, the GUI is no longer usable and must be closed.
Manager GUI outputs the following requests: save team configuration, change tactic, substitute player. It has to be able to receive the replies of those signals and the statistics of the players. The pattern is the same: the client part of the GUI issues asynchronous requests based on user inputs, the replies are received by the response subscriber of the GUI. The players statistic subscriber receive live notification exclusively during the simulation. As a design decision we prevent the Managers Users to play with the same team: once a team configuration

has been saved successfully on the Core that team is no longer available. During warm up state, the user is allowed to choose the team and to configure it (select line-up, formation, pressing level), then he/she saves the configuration. Assuming that the things go well, when the match is started the user can see the player statistics, change the team tactic and substitute team's players. On the other hand the GUI may receive a negative reply for the save configuration request in the following cases: if the user chooses the same team of the other Manager (the user must choose another team), if the match has already started in the meanwhile (the GUI passes to in-game state with correct settings), if the user made a subsequent save request after a previous one had already been accepted (the GUI passes to in-game state with correct settings). When the simulation ends, the GUI is no longer usable and must be closed.

9.1.5 GUI's termination

The user can decide at any time to exit from the GUIs. Whenever this happens, the subscribers of that particular GUI are forced to unregister from previously subscribed channels and terminate. An asynchronous call is made by the client part to make aware Billboard that the GUI has disconnected. The GUI frees up its resources and terminates.

9.2 Detailed design

Now we give an overview of internal multi-threading system of the GUIs. The main thread of the GUI (called event dispatching thread - edt), is responsible for carrying out the action requested by the user. The graphic components (buttons, combos, etc...) are registered to specific handler code (Listener). In this way the Listener (that runs on the edt) reacts to user actions executing the programmed instructions. If an action requires some time to execute, we must run it on another thread because otherwise further requests will hang (the edt keeps busy until it finishes executing the requested operation). We use different threads to model parallel activities. Such activities are the external input and output of the GUI. We define a separate thread that manages the external output, call it the client thread. Other threads are defined to deal with the external input of the GUI, they are the subscribers. One client thread is enough for our task, on the contrary we need more than one subscriber thread because we need to receive various type of concurrent notifications from Billboard.
We choose the Java language as the target language for the implementation of the GUIs. This choice guarantees a high degree of portability, in addiction it facilitates the development bringing a complete set of facilities for many different needs. We use the GUI widget toolkit (Swing) to build the graphics and layouts, the Workers Threads (SwingWorkers) to implement the GUIs internal multi-threading system, the XML DOM for the parsing of the xml configurations and the Yami4 facilities (an additional library) for the external communication. A package is defined for each GUI category.

9.2.1 Spectator GUI

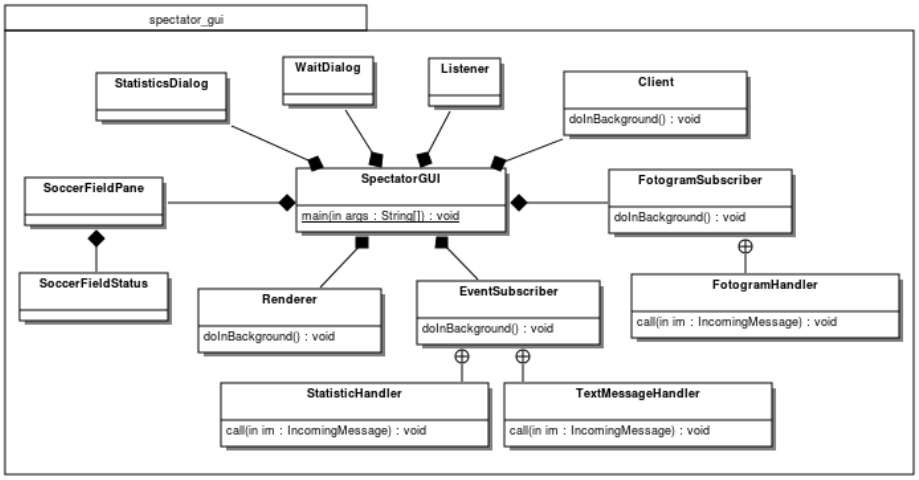

Figure 9.1: spectator_gui package

The application starts when **SpectatorGUI** main method is executed. This class is responsible for the creation and the termination of the instances of the other classes. The main method delegates the creation of the GUI to the event-dispatching thread (edt). Now the edt starts building the GUI: the Listener and the graphical components first. **Listener** is the class that contains the handling code for the events triggered by the user through the graphic components (buttons, combos, etc...). The Listener delegates the Client to manage the external requests. The graphic components are created and registered to the Listener. Some graphical components are enclosed in an own class: **WaitDialog** provide a simple modal dialog that is closed after a successful connection with the Billboard, **StatisticsDialog** provide a non-modal dialog that shows the statistics of the match, **SoccerFieldPane** provides the soccer field widget, it holds the graphic representations of the field, the players and the ball (the view). **SoccerFieldStatus** is a model for SoccerFieldPane, it holds the state of the soccer field widget (position of players within the field, position of the ball, etc...). The soccer field widget is modeled using an observer pattern: if we would move someone on the field, we must change the state of the widget, the view will change accordingly as soon as we repaint that widget.
After that the graphic layout has been created (but now shown yet), the edt proceeds on the creation of the threads: Client, FotogramSubscriber, EventSubscriber, Renderer. Only the Client thread is started, then the GUI is made visible. The **Client** holds the logic of the external output communication, it connects to Billboard input partition using the address and port given from command line. After successful connection, it starts the subscribers and remains active in order to be called by the Listener. The **FotogramSubscriber** and **EventSubscriber** hold the logic of the external input communication, they subscribe to the channels offered by Billboard output partition using the address and port given from command line. The callback happens on the call methods of their nested handler classes. FotogramSubscriber registers to the Fotogram Frame channel, the received callback updates the model of the field panel. The **Renderer** is started by the callback on the arrival of the first frame. Its only

concern is to repaint the soccer field widget at a regular interval. EventSubscriber registers to the Match Statistics channel and to the Messages channel. The statistics notification updates the statistics of the dialog, the message notification updates the commentary on the text area of the main window.

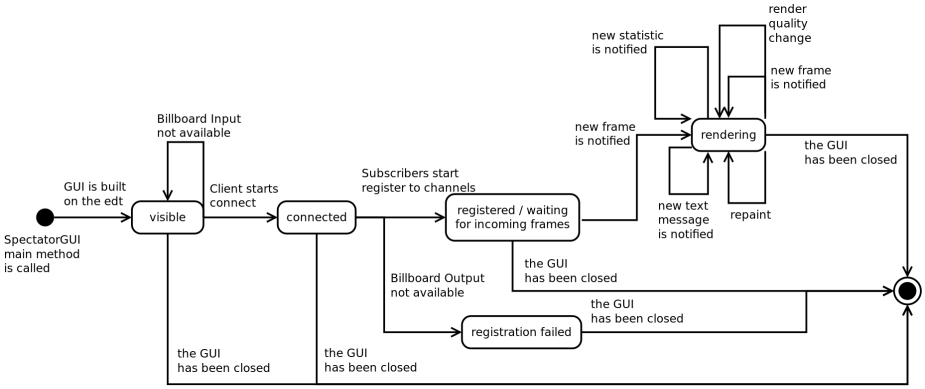

Figure 9.2: Spectator GUI state diagram

Whenever the rendering quality is changed, the FotogramSubscriber dynamically unregisters from the channel and registers to the new one, the Renderer receives the new refresh rate value.

9.2.2 Simulator GUI

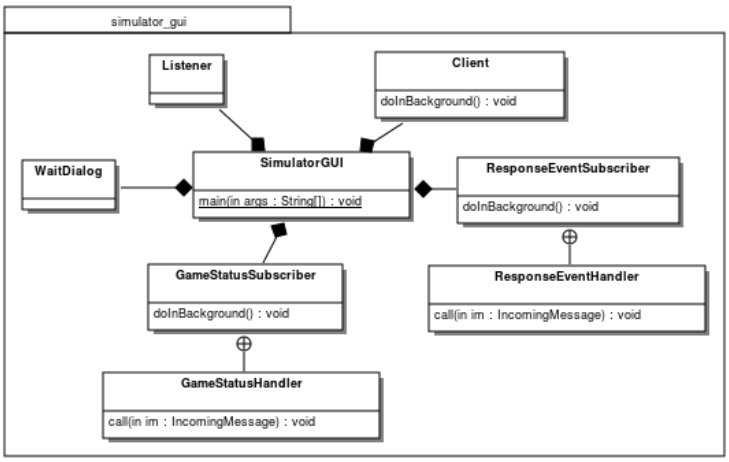

Figure 9.3: simulator_gui package

The application starts when **SimulatorGUI** main method is executed. This class is responsible for the creation and the termination of the instances of the other classes. The main method delegates the creation of the GUI to the event-dispatching thread (edt). Now the edt starts building the GUI: the Listener and the graphical components first. **Listener** is the class that contains the handling code for the events triggered by the user through the graphic components (buttons, combos, etc...). The Listener delegates the Client to manage

the external requests. The graphic components are created and registered to the Listener. **WaitDialog** provide a simple modal dialog that is closed after a successful connection with the Billboard.
After that the graphic layout has been created (but now shown yet), the edt proceeds on the creation of the threads: Client, ResponseEventSubscriber and GameStatusSubscriber. Only the Client thread is started, then the GUI is made visible. The **Client** holds the logic of the external output communication, it connects to Billboard input partition using the address and port given from command line. Client makes the synchronous connect call to Billboard, if the GUI is rejected, the subscribers are not started and the GUI needs to be closed. If the GUI is allowed, the Client starts the subscribers and remains active in order to be called by the Listener. The **ResponseEventSubscriber** and the **GameStatusSubscriber** hold the logic of the external input communication, they subscribe to the channels offered by Billboard output partition using the address and port given from command line. The callback happens on the call methods of their nested handler classes. The ResponseEventSubscriber receives from Billboard output the replies of the requests made by the Client part of the GUI. At the beginning it receives the connect reply that contains the initialization data: the string containing the simulation xml and the state of the match on the Core. With this information, after the parsing of the stringyfied xml file, it is able to trigger the appropriate state on the GUI.

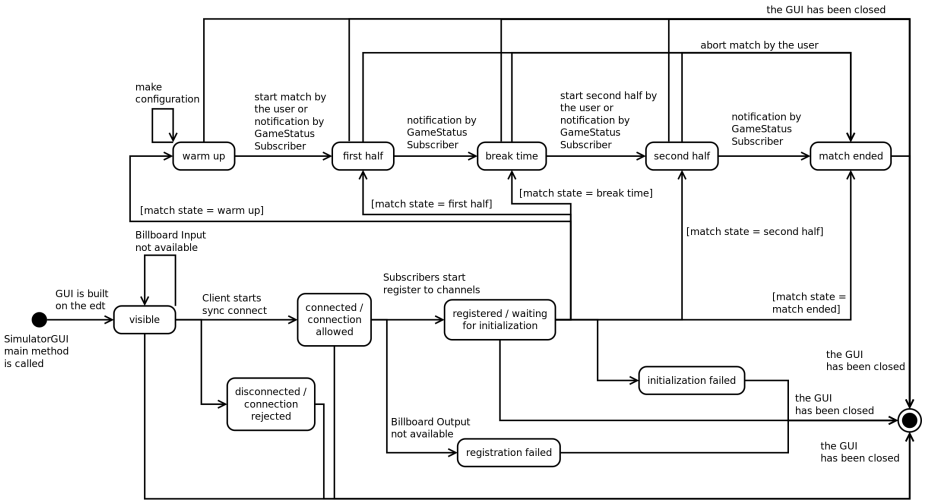

Figure 9.4: Simulator GUI state diagram

The other replies that it manages are, start match, start second half and abort match. The GameStatusSubscriber receives the notifications about the state of the simulation. Whenever it receive the new state, it triggers the appropriate state on the GUI. Actions on the state of the GUI are protected in order to avoid race conditions.

9.2.3 Manager GUI

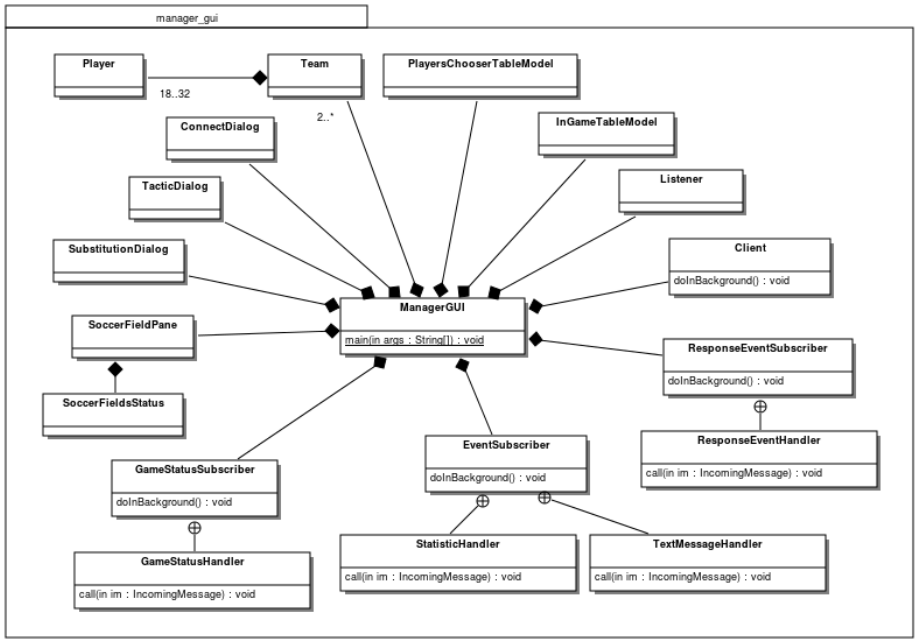

Figure 9.5: manager_gui package

The application starts when **ManagerGUI** main method is executed. This class is responsible for the creation and the termination of the instances of the other classes. The main method delegates the creation of the GUI to the event-dispatching thread (edt). Now the edt starts building the GUI: the Listener and the graphical components first. **Listener** is the class that contains the handling code for the events triggered by the user through the graphic components (buttons, combos, etc...). The Listener delegates the Client to manage the external requests. The graphic components are created and registered to the Listener. Some graphical components are enclosed in an own class: **ConnectDialog** provides a modal dialog used by the user to decide the role (locals/visitor) before connecting to the Billboard, **TacticDialog** provides a modal dialog that allows the change of the team's tactic (formation and pressing level) , **SubstitutionDialog** provides a modal dialog that allows the substitution of a player, **SoccerFieldPane** provides the soccer field widget, it holds the graphic representations of the field, the players (the view). **SoccerFieldStatus** is a model for SoccerFieldPane, it holds the state of the soccer field widget.
After that the graphic layout has been created (but now shown yet), the edt proceeds on the creation of the threads: Client, ResponseEventSubscriber, EventSubscriber and GameStatusSubscriber. Only the Client thread is started, then the GUI is made visible. The **Client** holds the logic of the external output communication, it connects to Billboard input partition using the address and port given from command line. The difference here is that we need to know the role (locals or visitor) before connecting, then the Client makes the synchronous connect call to Billboard. If the GUI is rejected, the subscribers are not started and the GUI needs to be closed. Once the GUI is allowed to cover the requested

role, the Client starts the subscribers and remains active in order to be called by the Listener. The **ResponseEventSubscriber**, the **EventSubscriber** and the **GameStatusSubscriber** hold the logic of the external input communication, they subscribe to the channels offered by Billboard output partition using the address and port given from command line. The callback happens on the call methods of their nested handler classes. ResponseEventSubscriber receives from Billboard output the replies of the requests made by the Client part of the GUI. At the beginning it receives the connect reply that contains the initialization data, the other replies are: save team configuration, change tactic and substitute player. The connect reply brings the string containing the teams xml and the state of the match on the Core. The stringyfied xml file is parsed and finally the thread populates the graphic structures along with their models triggering the appropriate state on the GUI. The **PlayerChooserTableModel** and the **InGameTableModel** are convenient classes that represent the models of the GUI tables. The PlayerChooserTableModel is used on the initial team configuration table and on the table of SubstitutionDialog. The InGameTableModel is used for the in-game table. The classes that stores the result of the parsing are **Team** and **Player**.

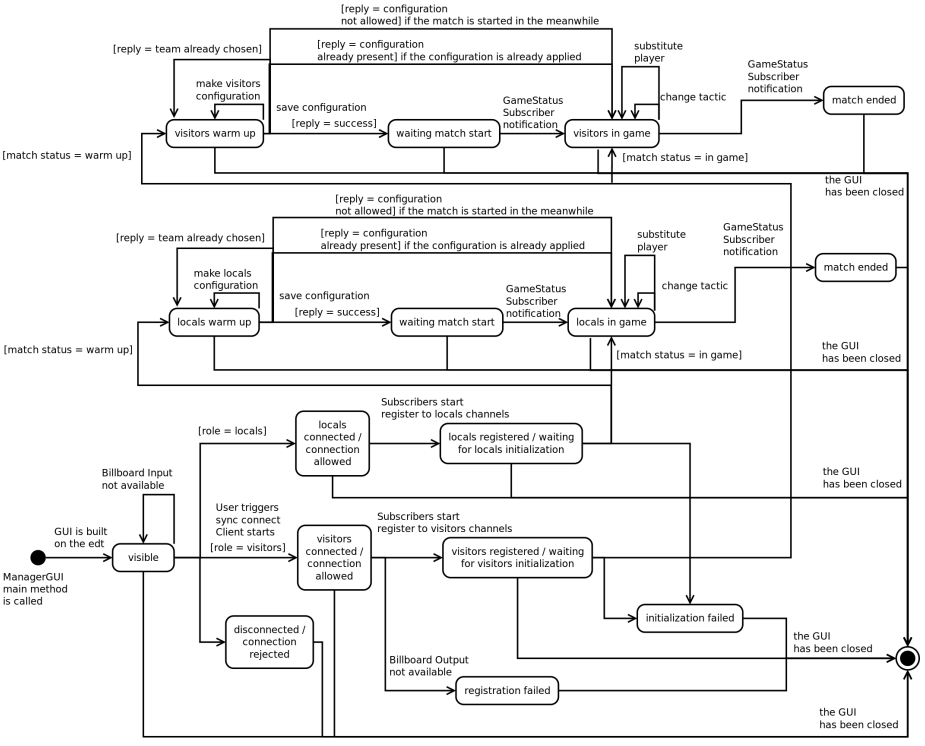

Figure 9.6: Manager GUI state diagram

EventSubscriber registers to the Player Statistics channel and to the Player Messages channel. The statistics notification updates the InGameTableModel, the message notification updates the player commentary on the text area of the main window. The GameStatusSubscriber receives the notifications about the state of the simulation, is used to trigger particular actions on the interface.

Chapter 10

Conclusions

This project consisted on the development of a soccer simulator. This was aimed at applying concurrency and distribution theory.
We learned that a number of guidelines exist to design a concurrent system. For example, the task architecture of a concurrent system is really important to develop a reliable system. As we already reported, tasks should not be active and servers at the same time, i.e. those that call other tasks should not receive calls. In practice this means that we need to design mechanisms that permit communication within tasks that need each other still avoiding potential deadlocks. At the same time, synchronization within tasks need to protect data structures, but also to develop reliable synchronization protocols. Finding the causes of misbehaviour due to concurrency design errors is not trivial.
There is also some room for improvements in the Ada design and implementation of our system. For example, the event-driven design that already exists in the core subsystem could be extended to completely cover it, then resulting in events that are generated from the very beginning (e.g. when a player attempts a tackle) and forwarded throughout the system.
Other improvements could be, for example, the intruduction of pause/resume features or the possibility of starting a new match without restarting the simulator when the previous match is finished.

Regarding distribution aspects, we can say that a big effort is needed in order to find suitable communication models and develop robust protocols for the interaction between the various components of a system. The most complicated work was to understand the different distribution vendors tools, because each vendor makes different and often non interoperable choices. We also had to establish an automatic mechanism to build and execute all the distributed subsystems. Other important choices regards load balancing. A well balanced system overcomes the limitations of a centralized system, the Billboard subsystem, for example, makes the whole system more scalable in that it allows the connection of an indefinite number of spectator without overloading the core subsystem. Additional aspects to care about are security, and availability, in our project this was not the main focus, but they are very important in many distributed systems to guarantees the consistency of the service.

An old american teacher said: *"The art of a benefactor is to take learners to the brink. A benefactor can only point the way and trick."* [17]
...AdaWay somehow tricked us.

Bibliography

[1] *Student programming contest "The Ada Way" - Specification*, (http://www.ada-europe.org/AdaWay/), Ada-Europe, 2010.

[2] *Ada Reference Manual*. Ada-Europe, 2005.

[3] Mordechai Ben-Ari, *Ada for Software Engineers*. Springer, 2nd Edition, 2009.

[4] Alan Burns, Andy Wellings, *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[5] *Laws of the Game 2010/2011*, Fédération Internationale de Football Association, 2010.

[6] Hassan Gomaa, *Designing Concurrent, Distributed and Real-time Applications with UML*. Addison-Wesley, 2000.

[7] Annarita Margiotta, *Studio del moto di una palla che rimbalza*, (http://ishtar.df.unibo.it/Uni/bo/scienze/all/margiotta/stuff/palla.htm), University of Bologna.

[8] Russell Miles, Kim Hamilton, *Learning UML 2.0*. O'Reilly Media, 2006.

[9] Andrew S. Tanenbaum, Marteen Van Steen, *Distributed Systems: Principles and Paradigms*. Pearson, 2nd Edition, 2006.

[10] Stuart Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd Edition, 2009.

[11] Markus Aleksy, Axel Forthaus, Martin Schader, *Implementing Distributed Systems with Java and CORBA*. Springer, 1st Edition, 2005.

[12] John Wiley and Sons, Chichester *Engineering distributed Objects*. Wolfgang Emmerich 2nd Edition, 2000.

[13] *YAMI4 - Messaging Solution for Distributed Systems*, (www.inspirel.com/yami4), Yami4, 2012.

[14] *OMG Data Distribution Service*, (http://www.rtcmagazine.com/articles/view/100285), Real-Time Publish/Subscribe Becomes a Standard, 2005.

[15] Eric Newcomer, Greg Lomow *Understanding SOA with Web Services*. Adisson Wesley, 1st Edition, 2005.

[16] *Polyorb mailing list*, (http://lists.adacore.com/pipermail/polyorb-users/2012-January/001226.html), Polyorb mailing list, January 2012.

[17] Carlos Castaneda *Journal to Ixtlan*. Washington square press 1st Edition, 1972.