Code-O-Soccer

User Manual

31st August 2015

Contents

1 Introduction

This is a framework that defines an API for writing Strategy for autonomously playing 2-wheeled Robot Soccer. This user manual aims to make the user familiar with the API to develop code that runs the robots to play robot soccer.

Code-O-Soccer API supplies a set of components that allows the user to write High-Level Strategy code and converts this to Low-Level commands for the robots. It also enables the user to implement coordination among robots playing soccer through use of a world model describing the robot states in the football field.

2 Installation and Setup

2.1 Linux

Currently, Code-O-Soccer API is not available for linux. However, we are still trying to get it up and running and will post updates if and when it is made available.

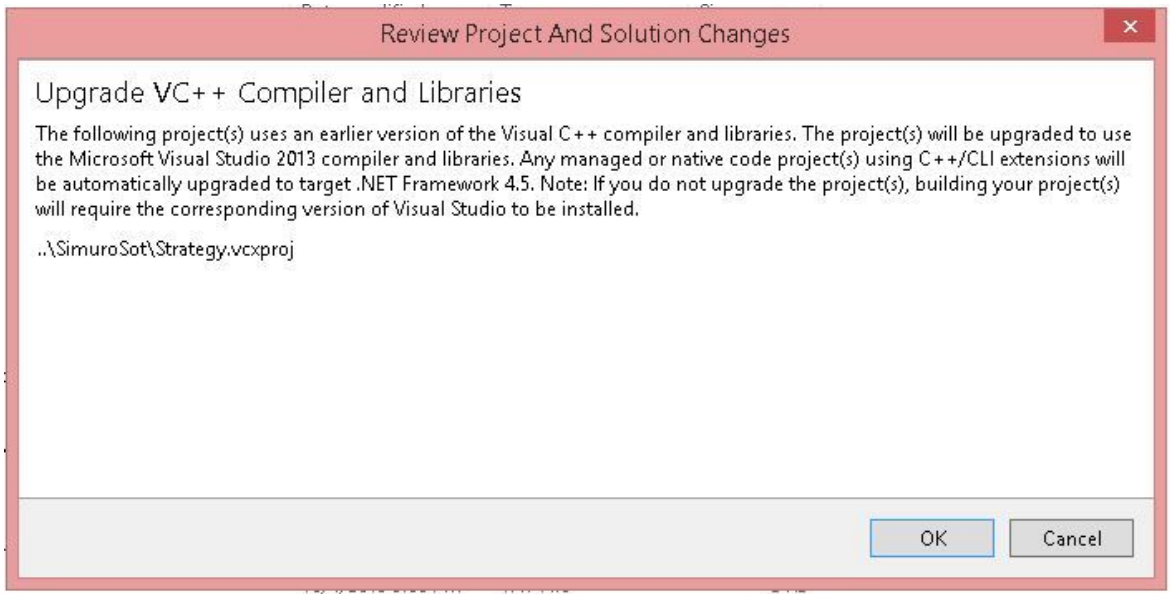
2.2 Windows

Visual Studio 2012 can be downloaded from http://go.microsoft.com/?linkid=9816768. To download the installer file Go to http://event.krssg.in

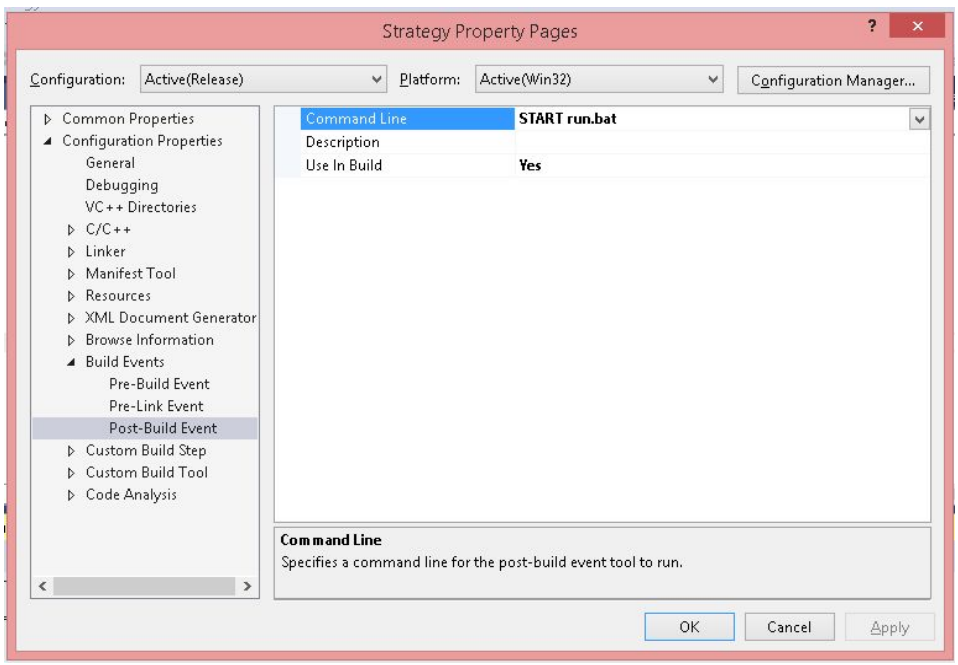
Setting up the project:

Setting up Debugger and project:

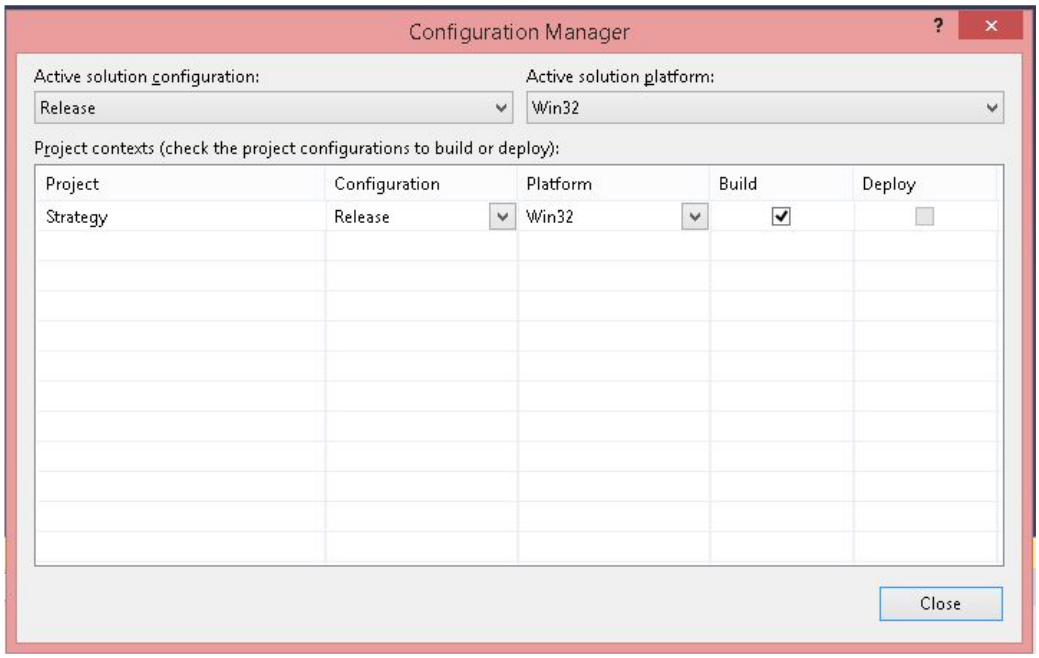
1. Copy “SimuroSot” to C:\. Drive.
2. Copy the Strategy folder to C:\. Drive.
3. Open “ConsoleDebugger.sln” from C:\SimuroSot\ConsoleDebugger.
4. Select ok if prompted to upgrade.



5. Choose Build from build menu.
6. Exit the project and open “Strategy.sln” from C:\SimuroSot.
7. Go to Project->Strategy Properties->Configuration Properties -> Build Events -> Post-Build Events
8. Add (without quotes) "START run.bat" to command line field.



9. Choose configuration manager from Build menu
10. Choose Configuration Release mode.



11. Choose “Build Solution” from Build menu.
12. Use “Run.bat” after building your solution from C:\SimuroSot to further start the simulator

Start Coding your Strategy !

Running Simulator

1. To change your Team color: To change your team color, write your team color BLUE_TEAM/YELLOW_TEAM against teamColor field in game.hpp near the top. Make sure to choose the corresponding team color in the debugger and simulator as well.

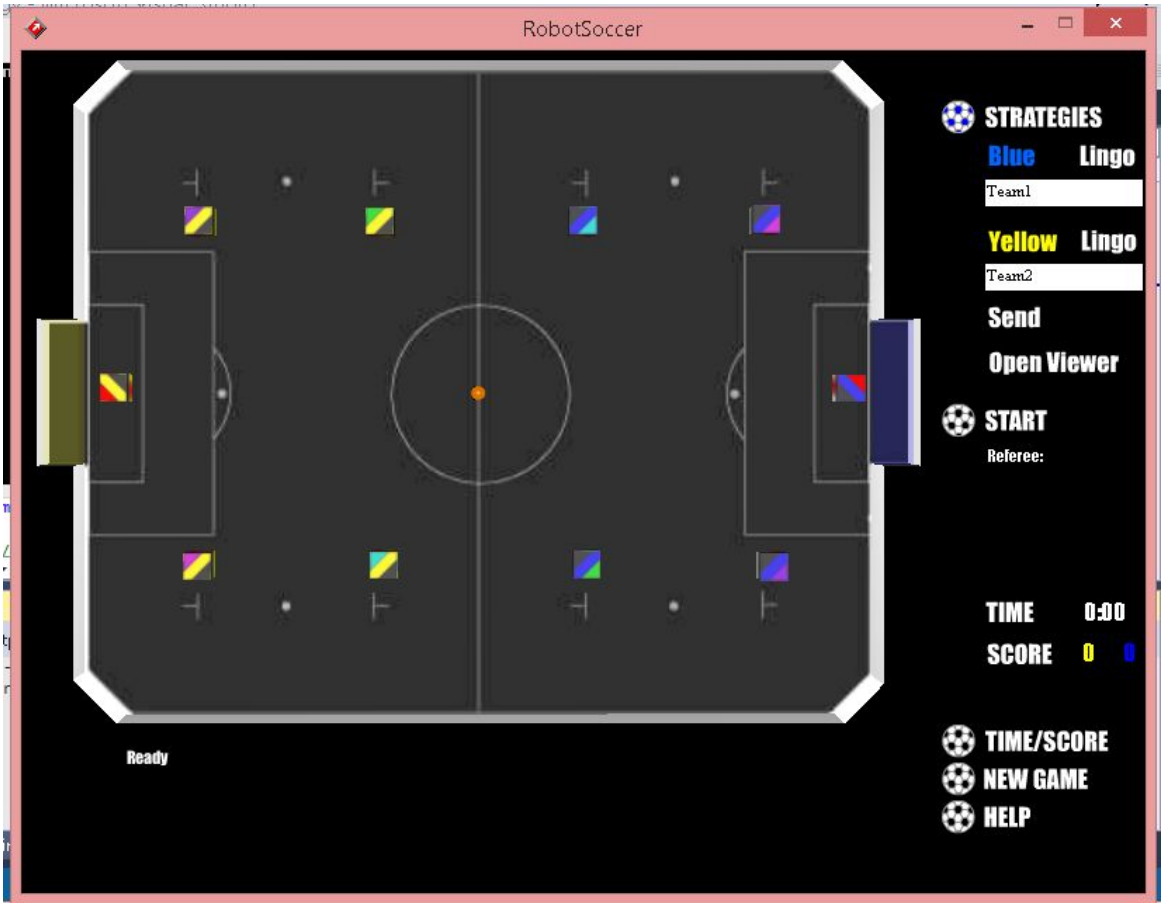
Game.hpp

(Global Scope)

```
// For adding header files define your includes here and add the headers in Game folder.
// For example, You may see these files - Attacker.hpp,Defender.hpp and Goalkeeper.hpp
// For checking out the skills you may see skills.h placed in Skills folder.
#pragma once
#include "skills.h"
#include "Attacker.hpp"
#include "Defender.hpp"
#include "GoalKeeper.hpp"

// Change your team color here (BLUE_TEAM/YELLOW_TEAM)
Simulator::TeamColor teamColor = Simulator::BLUE_TEAM;

// Make usingDebugger is false when playing against an opponent
bool usingDebugger = true;

namespace MyStrategy
{
  // Write your strategy here in game function.
  // You can also make new functions and call them from game function.
  void game(BeliefState *state)
  {
      attacker(state,2);
```

100 %

2. Press F5 or choose Debug->Start Debugging.
3. Choose STRATEGIES from right panel. Following windows will appear.



4. Toggle between C++/ Lingo by clicking on it against specific team.
5. Choose START

3 The Robots

The robots used, both in the simulator and real-world matches, are 2-wheeled differential drive robots. This means that the robot has 2 degrees of freedom.

Fig 1 shows the robots used in the Simulator. The center-color identifies the team of the robot,

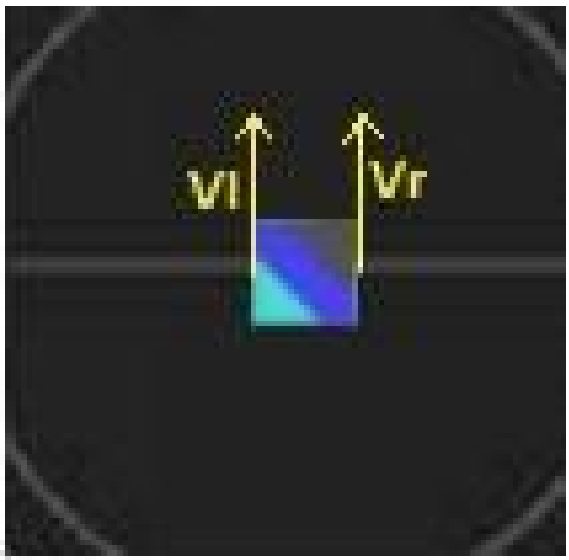


Figure 1: Top View of Simulator Robot

either blue or yellow. The rest colors form a pattern, which is distinct for each robot and is used by the vision module for identification of robots.

Robot controls for translation velocity, angular velocity are all provided in the API. These will be explained alongside skills in more detail. Fig 2 shows the actual 2-wheel robot.

4 Architecture

The functioning of a generic program created through the CodeOSoccer API is shown in Fig 3. The Vision Module reads from a camera and provides robot locations. These are sent to the Belief State module which populates a world model. This data is provided to the Strategy code. The

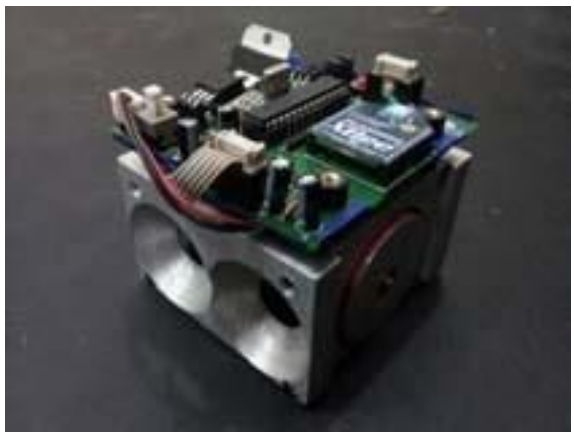


Figure 2: 2-Wheel Robot

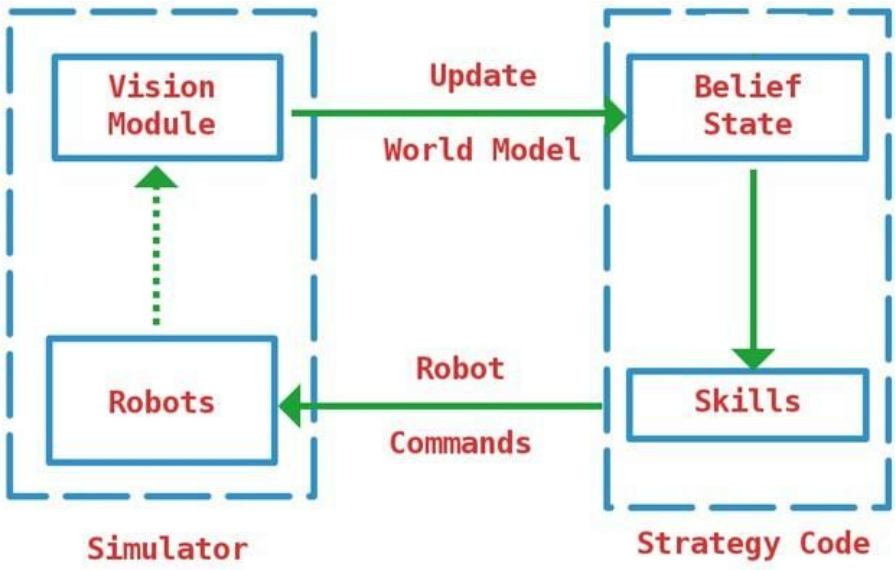


Figure 3: Code Architecture

Skills and Tactics layers are high level abstractions for controlling the robots. They output lowlevel commands to be executed by the on-board microcontrollers of the robots. These commands are communicated wirelessly to the robots.

In the Code-O-Soccer API, source code to most of these abstraction layers (eg. Vision module, Skills) is hidden from the user. Furthermore, the Vision Module and actual robots are replaced by a **Simulator** to allow the user to test his code. The user only needs to interact with the **Skills** and the **Belief State** world model to write high-level strategies. The same code, with some minor changes, may also be run on the actual robots. The following sections explain the Skill Layer and the Belief State world model, and how the user must write code which complies with the API to run the robots on the simulator.

5 Getting Started with Skills

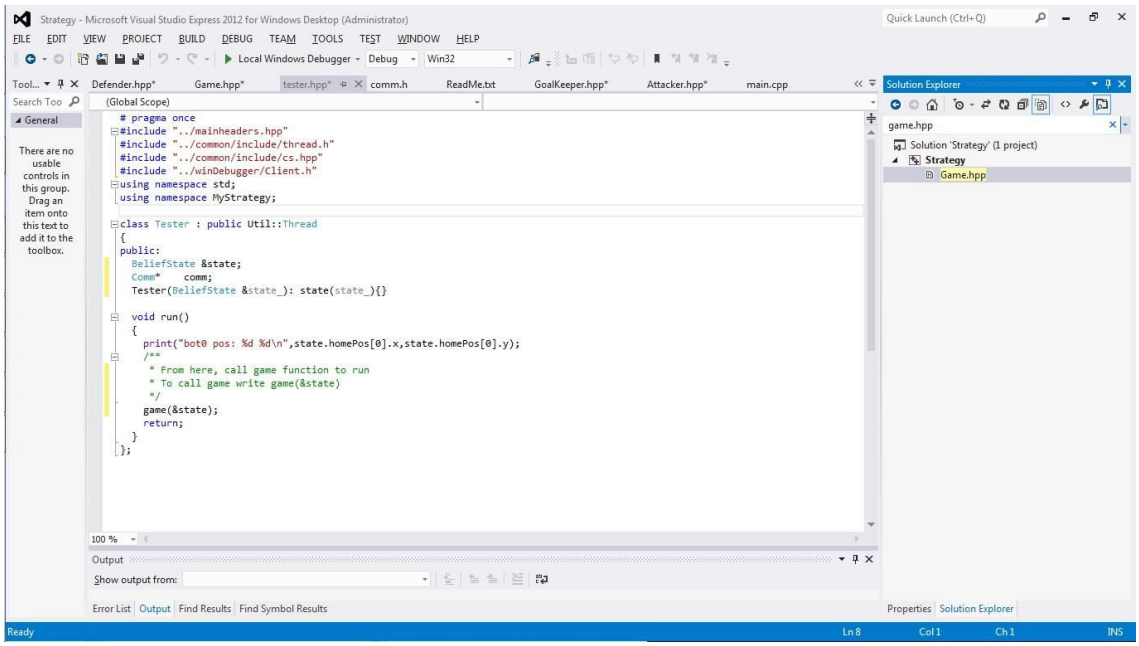
5.1 Introduction

Skills are the lowermost level of single-robot control. Each skill encapsulates a single-robot behavior. Each skill is parameterized, allowing for more general skills to be created which are applicable to a wider range of world states. For example, a "**Velocity**" skill would have speeds of the left and right wheels as parameters.
Some skills are simple in functioning (eg. "Velocity" skill simply gives the velocity to the left and right wheels of the robot) while others have more complex functions (eg. "GoToPoint" makes the robot move toward the given point with path planning).

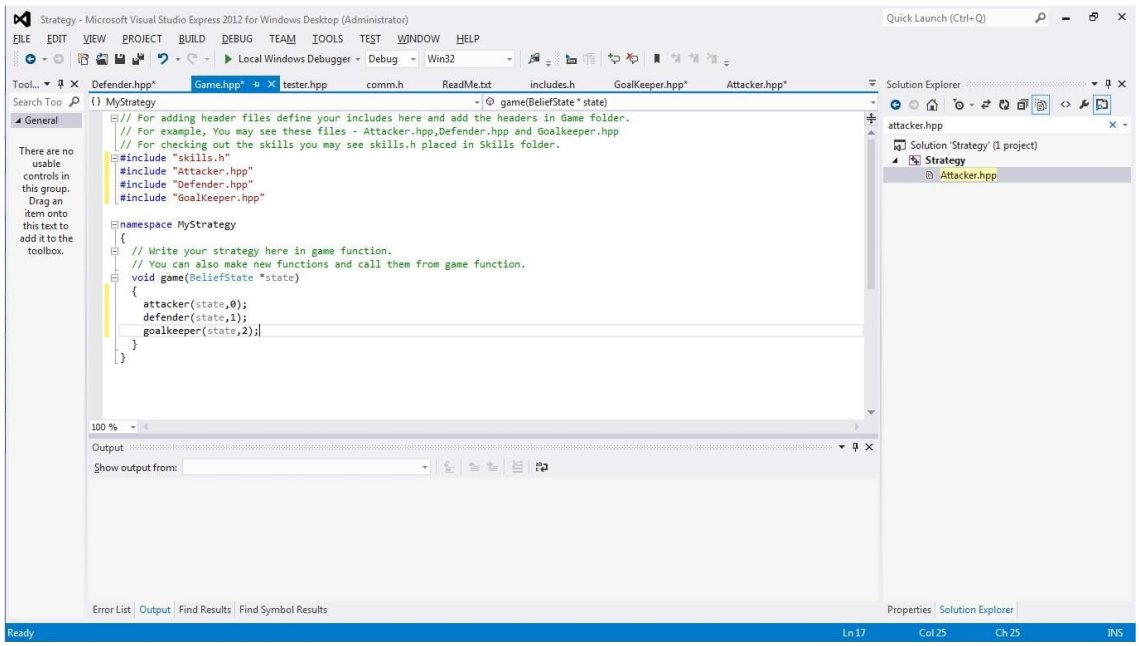
As mentioned before, each skill has its own parameter. For example, the **"Velocity"** tactic uses the parameters **botID**(id of the bot), **vl**(velocity of the left wheel) and **vr**(velocity of the right wheel). The header file **"skills.h"** describes each skill and their parameters. Please refer to the documentation for the complete list of tactic parameters.

5.2 Skill Usage

We now describe how to use skills.
Open the Visual Studio Solution file (Strategy.sln in Simurosot folder) in Visual Studio 2012. The Strategy project will be set as active.



The game function call is necessary to start the game. The parameter **State** is a pointer type variable of **BeliefState** From the solution explorer on the right-hand-side, open the file **game.hpp**.



The void **game(BeliefState* state)** function is the primary function to be used. The function is called repeatedly from **tester.hpp**. All the user code for robot control should be called in this function. We now go into a line-by-line analysis of the function.

```
void game( BeliefState *state) {
```

BeliefState is a class that defines the world model of the football field. An object of this class contains information about the positions, orientations and velocity of all the robots, position and velocity of the ball, field configuration etc.. It also contains some **predicates**, which identify the game situation. These will be discussed in detail in Section 6. **attacker**, **defender**, **goalkeeper** are the functions for different roles of the bot.

```
attacker(state,0);
```

This statement **executes** attack on a robot. 1st parameter is the BeliefState* argument. 2nd parameter is the bot ID (in this case bot 0). The user should be careful in passing the appropriate parametes.

The rest of the statements in the function have similar functionality, except defender and goalkeeper functions have been called. Similarly, other functions may be called from. For example, the following simple code would make bot 0 move forward:

```
void game( BeliefState *state) {
        Velocity(0,50,50);
        //Velocity skill being executed on robot 0 vl = 50, vr = 50
}
```

5.3 Make your own Skill!

To make a new skill, the user may define a function for the skill in **skills.h**. And whenever the user wants to use this function, the function can be called from that file.

Now we shall look into the BeliefState class and how to use its **"predicates"**.

6 Belief State

6.1 Introduction

As mentioned before, **Belief State** defines the world model of the football field. It contains information about the positions, orientations and velocity of all the robots, position and velocity of the ball, field configuration etc.. These will be called **State Parameters**. Additionally, it contains certain *"perceived"* information of the game: which team is in possession of the ball, is in attack, is in a particular role etc.. These are called **High-Level Predicates**.

6.2 State Parameters

Table 6.2 shows a list of state parameters present in BeliefState class and their use.

State Param		**Description**
Point2D<int>	homePos[HomeTeam::SIZE]	Array which stores home team member's positions.
Point2D<int>	fieldCentre	Stores field centre coordinates.
Vector2D<float>	homeVel[HomeTeam::SIZE]	Array which holds our bots' velocity components
Vector2D<float>	homeAcc[HomeTeam::SIZE]	Array which holds our bots' acceleration components.
float	homeAngle[HomeTeam::SIZE]	Array which holds angle of each bot with respect to positive x axis.
float	homeOmega[HomeTeam::SIZE]	Array which holds our team members' angular velocity.
float	homeAngAcc[HomeTeam::SIZE]	Array which holds our team member's angular acceleration.
Point2D<int>	awayPos[HomeTeam::SIZE]	Array which stores opponent team member's positions.
Vector2D<float>	awayVel[HomeTeam::SIZE]	Array to hold opponent bot's velocity components.
Vector2D<float>	awayAcc[HomeTeam::SIZE]	Array to hold opponent bot's acceleration components.
float	awayAngle[HomeTeam::SIZE]	Array which holds angle of each opponent bot with respect to positive x axis.

float	awayOmega[HomeTeam::SIZE]	Array to hold opponents team member's angular velocity.
float	awayAngAcc[HomeTeam::SIZE]	Array to hold opponents team member's angular acceleration.
Point2D<int>	ballPos	Holds position of ball at that time.
Vector2D<float>	ballVel	Holds velocity of ball at that time.
Vector2D<float>	ballAcc	Holds accleration of ball at that time.

The game function is passed a BeliefState object **state**. This BeliefState object is continually updated in the background. The State Parameters can be accessed through this object. For example, (x,y) coordinate of the 2nd (0 base indexing) robot of opponent team can be accessed from **state->awayPos[2]**.

6.3 Predicates

We briefly describe **High Level Predicates**.

6.3.1 High level Predicates

Table 6.3.1 shows a list of high-level predicates.

Predicate		**Description**
int	ourBotNearestToBall	Holds botid of our bot nearest to ball.
int	oppBotNearestToBall	Holds botid of opponent bot nearest to ball.
bool	pr_ball_in_our_dbox	True when ball in our D box. Computed in computeBallInDBox().
bool	pr_ball_in_opp_dbox	True when ball in opponent D box. Computed in computeBallInDBox().
bool	pr_oppBall	True when ball is with opponent.
bool	pr_ourBall	True when ball is with us.
bool	pr_looseBall	True when no one possesses the ball.
bool	pr_ballOppSide	True when ball is in opponent half.
bool	pr_ballOurSide	True when ball in our half.
bool	pr_ballMidField	True when ball is in region defined as midfield.
bool	pr_ballInOurCorner	True when ball in our corner .
bool	pr_ballInOppCorner	Predicate true when ball in opponent corner.

All predicates (with the exception of ourBotNearestToBall and oppBotNearestToBall) are **boolean** variables. Again, the BeliefState object passed to run_bot is continually updated in the background, and hence the predicates are being re-evaluated in the background. The following is an example of using predicates to change the tactic executed by robot 0:

```
void game(BeliefState *state)
{ if(state->pr_ballOurSide) {
                    Vec2D destinationPoint(OPP_GOAL_X,0);
                GoToPoint(0,state,destinationPoint,0,true,true); } else
                GoToBall(0,state,true);
}
```

When the ball is in our possession, robot 0 is instructed to go towards the opponent goal. **destinationPoint** has the coordinates of the midpoint of the opponent goal(**OPP_GOAL_X** stores the x-coordinate of the opponent goal). The fourth parameter is **finalslope** (the final angle that the bot should align). Last two parameters are

respectively **increaseSpeed**(whether bot should accelerate while moving) and **shouldAlign** (decides whether the bot should align to the **finalslope**.
If **shouldAlign** is **false**, **finalslope** will have no effect).

If the ball is not in our possession then robot 0 is instructed to go to ball.

Like state parameters, these predicates can be accessed through the BeliefState object passed to game.

7 Conclusion

We have provided a detailed description with working example of building code in the Code-OSoccer API. A complete knowledge of all that is covered in this manual is not necessarily required by the user to build his/her code. We wish the user an enjoyable experience in coding for CodeO-Soccer!