Behavior

Man-Hon Choi and Minguo Zhao[1]

[1] Tsinghua University, Beijing 100084, China
mgzhao@mail.tsinghua.edu.cn

Abstract. Introduction of Team Hephaestus Behavior module.

Keywords: XABSL

1 Brief Introduction

Based on the ROS Kinetic robot software platform on Ubuntu 16.04, we have written a decision node to deal with the behavior that the robot should take in different states.

The decision node needs to receive information about the position of the ball, goal and obstacles from the visual node, the position and orientation of the robot on the field from the positioning node, the information about the game status from the game controller node, the angular state information (yaw, pitch) of the head node and the information about the robot's motion state from the gait node. According to these information, after calculation and processing of the decision node, corresponding decisions are made, and movement commands are issued to the head node and the gait node, so that the robot can make corresponding actions in the game.

The program of the decision node is composed of XABSL(Extensible Agent Behavior Specification Language) [1] code and C++ code. The finite state machine is written by XABSL, so the decision is hierarchical. The XABSL program is mainly responsible for the transition of decision states, and also points out the actions to be taken in different states, but ultimately all basic behaviors rely on C ++ for specific implementation.

The advantage of using the XABSL program to implement the finite state machine is that it is hierarchical, the concept of each state is very clear. Each OPTION(program file) can be regarded as a module, which can be called when needed. Each state consists of two parts, DECISION(program segment) and ACTION(program segment). The state transition conditions and state behavior are clear and distinct. According to the state machine structure view in XABSL, the relationship between states can be clear. The implementation of all basic behaviors is left to C ++, and XABSL is essentially the upper-level caller and is responsible for the upper-level decision logic, so the code is highly readable.

2 Algorithm

The decision of our striker robot is different from the goalkeeper's. So we will introduce these two different decisions separately.

2.1 Striker

Our first layer of the decisions for striker is called opt_game. It is mainly responsible for processing the game state, and its structure is shown in Figure 1.

When the program start to run, it firstly enters the Initial_state in the first layer. The robot's behavior in this state is to look up and stay still. After receiving the ready message from the game controller, it jumps to the Ready_state. The robot's behavior in this state is to walk into the field and stop at a suitable location. After receiving the start message, it jumps to the Start_state. This state calls the second layer of the decision, namely opt_attacker_loc_vis, which is the core of the decisions of the striker robot in the game. All states in opt_attacker_loc_vis are actually considered as sub-states of the Start_state. Of course, there are other states in the first layer of decisions, such as Set_state and End_state, but these states rarely call the next layer of decisions, so we won't go into details.

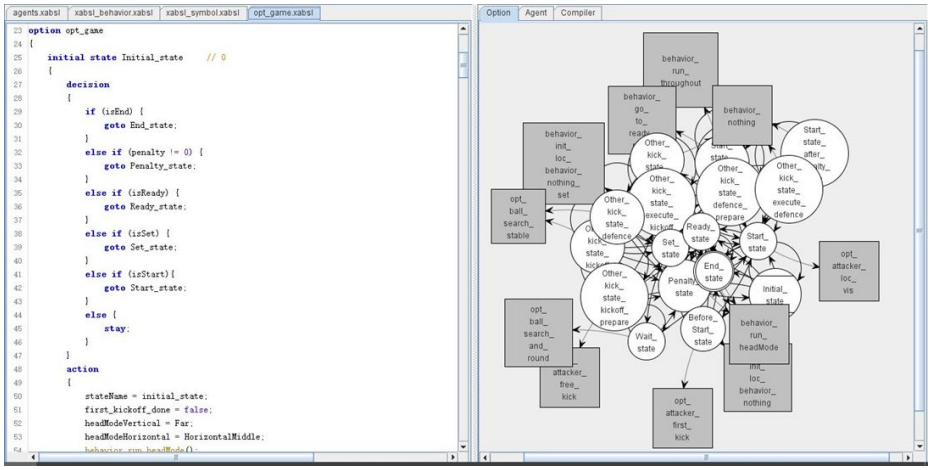

Fig. 1. The structure of the first layer of decisions is shown in the right side and the corresponding XABSL code is shown in the left side.

In the second layer, the main process is as follows: first of all, its initial state is search_ball state. The robot will first search for the ball in situ. If it cannot find the ball, it will travel throughout the playing field. If the ball is found, it will jump to the close_to_ball state. Then the robot will walk towards the ball and adjust the orientation of its body so that it faces the ball while walking. When the robot is less than 0.5 meter away from the ball, it will jump to other states. (Some complex actions are achieved by invoking lower-layer decisions)

If the robot is in its own half at that time, the program jumps to adjust_for_goal_loc state. In this state, the robot will use the positioning information from localization node to adjust the orientation around the ball until it is adjusted to the direction suitable for kicking (Generally facing the opponent's goal), then enter kick_ball state and kick the ball out.

If the robot is in the opponent's half at that time, the robot first looks up and stops. The program jumps to search_goal state which uses computer vision to find the goal. In this state, the robot will first try to find the goal in situ. If it can't find the goal, it rotates an angle around the ball and then continues to look for the goal. When the goal is found, enter a state called adjust_for_goal_vis to adjust the kick direction according to the known goal position. Then it enters kick_ball state and kick the ball out.

When the ball was kicked out, the program jumps back to the search_ball state. So this is the main process of the decisions for our striker. Of course there are many complicated branches, but because it is too trivial, we won't go into details.

2.2 Goalkeeper

The first layer of the decisions for our goalkeeper is similar to striker's, but the second layer is much different.

After receiving the start message, the program jumps to the Start_state. This state calls the second layer of the decision, namely opt_goalkeeper_attacker, which is the core of the decisions of the goalkeeper robot in the game. Its structure is shown in Figure 2.

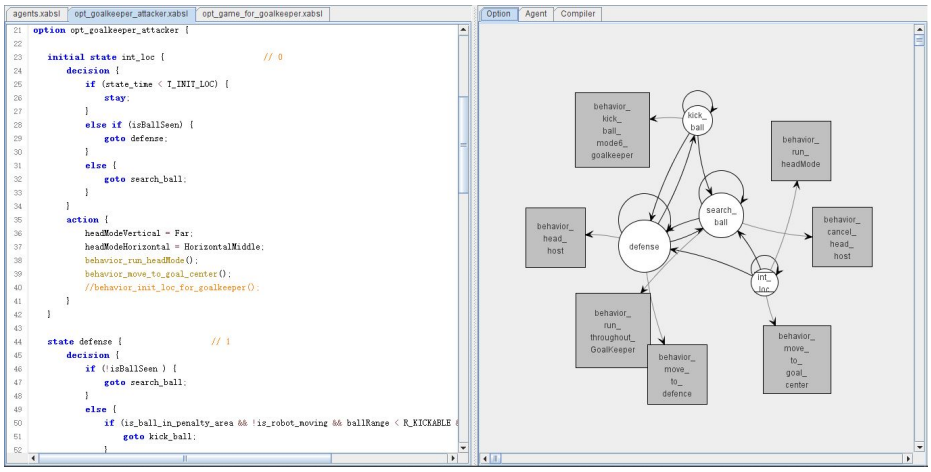

Fig. 2. The structure of the second layer of the decisions for goalkeeper is shown in the right side and the corresponding XABSL code is shown in the left side.

The initial state is int_loc. The robot looks up and moves to the center of the penalty area of its own half. In the initial state, if the robot does not see the ball after 1 second, it will jump to the search_ball state. The robot does not move its feet in place,

and only turns the head to search the ball. If the robot find the ball in the initial state or search_ball state, it will jump to the defense state. Then the robot will make corresponding actions according to the position of the ball. For example, moving itself to block the way of the ball.

In the defense state, if the robot can no longer see the ball, it will jump to the search_ball state. If the ball is already in the penalty area and the robot has moved directly behind the ball and stopped, the program will jump to the kick_ball state, the robot kicks the ball out of the penalty area, and then the program returns to the defense state.

3 Conclusion

We use XABSL to implement the finite state machine and make the program highly readable. XABSL is essentially the upper-level caller and the implementation of all basic behaviors is left to C ++. We have introduced the main decisions of our striker and goalkeeper robots. The main process is simple, but the decision node actually contains many complicated branches to deal with many special cases.

References

1. Loetzsch, Martin & Risler, Max & Jüngel, Matthias. (2006). XABSL - A pragmatic approach to behavior engineering. 5124-5129. 10.1109/IROS.2006.282605.