Description of the Team Virtual Werder 3D 2004

Tjorben Bogon, Mirco Kuhlmann, Cord Niehaus, Steffen Planthaber, Carsten Rachuy, Arne Stahlbock, Ubbo Visser, and Tobias Warden

TZI - Center for Computing Technologies, University of Bremen, Germany
grp-vw3d@tzi.de
WWW home page: http://www.virtualwerder.de

Abstract. Virtual Werder 3D is a new team, which has been developed based on the test agent given in the package. Right now, the team has the same architecture but a different behavior than the simple client. This includes a new formation method and some additional basic functions such as *slow down, positioning, and move around ball* in the play_on mode. Future work includes the integration of planning and plan-recognition components as well as other MAS components such as a deliberation module and reactive components.

1 Introduction

Programming robots in dynamic environments under real-time conditions requires heterogenous information from various sensors or sources. The majority of the sensor data are quantitative. However, we identified qualitative knowledge to be also crucial, especially in environments where time is an issue. One part of this qualitative knowledge is spatial knowledge.

We intend to integrate our team Virtual Werder 3D into our general multi-agent model which is based on both quantitative and qualitative information. We will use a formal, egocentric, and qualitative approach to navigation which overcomes some problems of quantitative, allocentric approaches. By the use of ordering information, i.e., based on a description of how landmarks can shift and switch, we generate an *extended panoramic representation*. Since our approach abstracts from quantitative or metrical detail in order to introduce a stable qualitative representation between the raw sensor data and the final application, it can for example be used in addition to the well-elaborated quantitative methods.

The following sections are organized as follows: firstly, we will give insight into the new basic functions (skills) that we have implemented so far. These skills are necessary for our approach that we will draw in the succeeding sections. We will describe our future architecture with its components.

2 Basic implementation

The following implementation details form one part of the basic functions that we need for our overall approach. The next subsections give an overview about the development and changes that we have done so far. This includes a development before kickoff and behavior changes while the play is on.

2.1 Formation

Our agent implements three functions which are concerned with the position of the agents in the general team lineup.

The method *getHomePosition()* acts as a kind of control function which delegates the actual task of calculation of the correct home position of any agent to two other function explained further below according to the game state. Each of these functions return a position for the current agent based on its tricot number (e.g. agent no. 1 is the goal keeper).

Before kickoff the function *getStartUpPosition()* returns a startup position which is characterized by the fact, that no matter which tricot number has been assigned to the current agent the position can be located somewhere within the own half of the field. This means that the *startupPosition* is somewhat defensive. When the game is in another state the function of choice is *getLineUpPosition()* which returns the position of the agent in the actual team lineup (e.g. goal, defense, mid-field, forward).

At the moment we only implemented the *getHomePosition()* function to be called before kickoff with the result that all agents are set on non-standard positions in regard to the test-agent.

2.2 Basic skills

The following methods have been implemented:

Slow down: We implemented a function which slows down the agent when approaching a given destination point. Using this function before kicking the ball the agent slows down which prevents him from being to fast and therefore stumbling across the ball.

Moving around the ball: This function enables the agent to move around the ball. It is used if the kickoff-position lies "behind" the ball (relative to the agent). This results in the agent making a circle around the ball instead of running against / stumbling across it.

Positioning: This function checks whether the agent is in the correct position to kick the ball into the opponents goal[1][2] For this calculation the agents uses the vectors pointing towards the left goal-flag, the right goal-flag and the ball, relative to the agent himself. This results in the agent kicking the ball only if the goal can be hit which is significantly better than just kicking if the agents distance toward the ball is below the minimal kick-distance.

[1] Due to the fact that the agents start on the opponent's side of the game-field and we did not notice this on time our agent's behavior is somehow "flipped". That means that the agents are playing towards their own goal and not the opponent one.

[2] Our agent uses (by now) the myPos-Value, therefore it has to be enabled for our team.

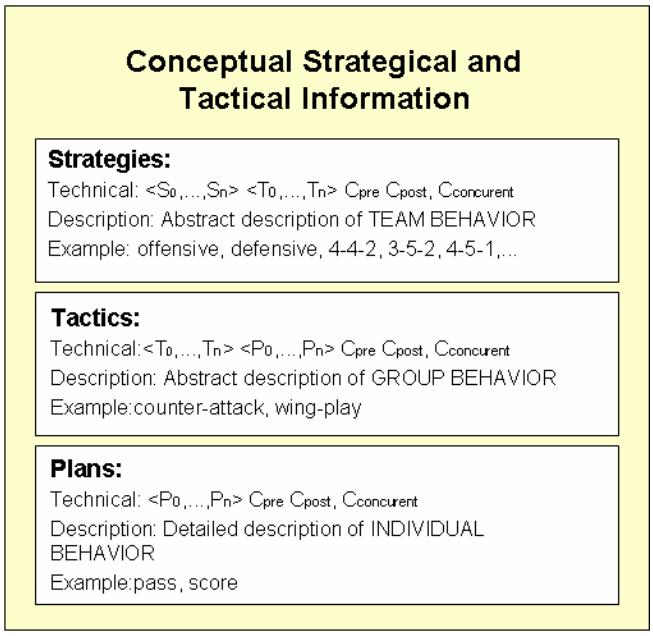

Fig. 1. The three knowledge levels

3 Architecture

The architecture of VW3D can be distinguished in two levels: the knowledge level and the processing level. The knowledge level describes the knowledge we need to describe strategies, tactics, and plans in the soccer domain. Please note, that these levels could vary in another domain. The processing level describe processes that operate with this kind of knowledge to derive concrete plans for the agents.

3.1 Knowledge level

We follow the intention to formalize soccer knowledge from an experts point of view. We consider three parts in the knowledge level (cf. figure 1) which are the following:

- Strategies
- Abstract tactics
- Concrete tactics/concrete plans

The strategic level includes knowledge about major strategies such as offensive or defensive play, formation etc.. It is the highest level and knowledge form this level will be used for the whole team only a few times during a play (e.g. 4-4-2 formation).

The next level is the level of abstract tactics which are dependent upon the strategies. This is the level of group behavior where we can see tactics about certain defenses (e.g. last defense line) or certain wing counter-attacks. These tactics are translation-invariant which means that certain tactics can be applied regardless of the position of the group members. An example for this is the offside trap, which can be applied anywhere on the field. An abstract tactic also includes potential roles of players. If we would have a fast left-wing counter-attack for four players, for instance, we could play this counter-attack with one defender, two mid-fielder, and one attacker. This however is only one variant, the actual instantiation might look different on the concrete plan level.

The concrete tactics level is the level where the variants of the abstract tactics are being solved. One example for this is a fast counter-attack (left or right). We will have two branches with concrete instantiations for every abstract role description. One example might help: suppose we have one strategy and one abstract tactic (fast counter-attack). We could have three different role descriptions for this abstract tactic with two or three concrete instantiations. This results to nine concrete combinations for the game. We can play left or right with each of the nine variants, i.e., a simple counter-attack results in 18 different variants.

The goal is a graph which can be used for execution.

3.2 Processing level

The following figure 2 shows the modules that are necessary to implement the mentioned features. These modules are basically the same for each knowledge level, only the outcome of each module varies with respect to the knowledge level.

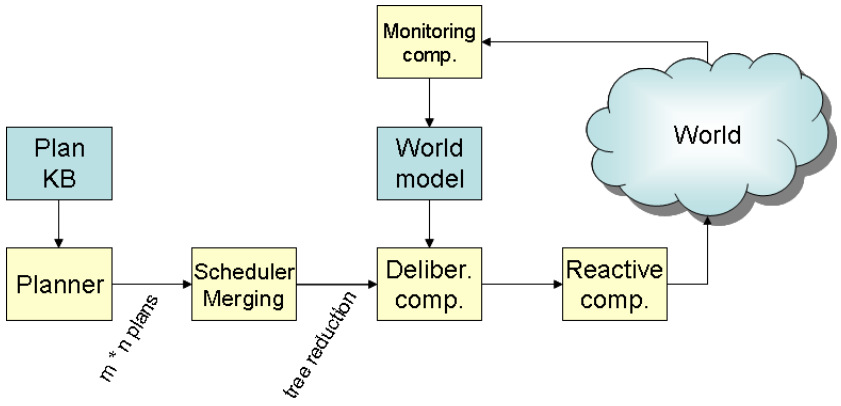

Fig. 2. Processing levels of VW3D

The planning module uses the knowledge that is contained in the plan knowledge base to generate plans. Suppose we are on an abstract tactics level. The

planning module would generate, e.g. three slow counter-attacks and four fast counter-attacks based on different role instantiations. A merging component that is included in the scheduler, takes those plans and identifies possible merging opportunities. The purpose is to reduce the number of plans and therefore complexity. The deliberative component takes this outcome and selects between the possible plan variants. The outcome in our example might be the second fast counter-attack. The reactive component takes this plan and executes it.