The Dainamite 2006 Team Description

Holger Endert

DAI-Labor, TU Berlin
Faculty of Electrical Engineering
and Computer Science
holger.endert@dai-labor.de

Abstract. This paper gives an overview about the structure of the *Dainamite* robocup team, which was implemented and used for teaching since october 2004. The focus of this document lies on the architecture of an agent, additionally highlighting some of its features like action selection and *Situation Based Perception*.

1 Introduction

The *Dainamite* robocup team was implemented and improved during student projects since october 2004 [1]. Much of the work done was influenced by [1], which provided a good guideline for developing the the basic skills, the world-model and the synchronization with the server. However, the most important skills like passing and ball interception where replaced by optimized planning methods, so called situations, and the tactic component, which decides what action to perform, was built upon own ideas. This paper describes the *Dainamite* architecture by firstly giving an overview about the constituting components of an agent, followed by the description of the main components, the synchronization, the tactic (or action selection) and the skills. Finally closing with a discussion of the advantages and weaknesses of this approach.

2 The Agent Framework

Dainamite agents are made up of six constituting parts, which are the world-model, the perception, the synchronization (short: synchro), the action, the tactic and the planning devices. Its behaviour results mainly from the interaction of these componenents. In Fig. 1, their structure and control flow within the agent is visualized. Due to its reactive nature, acting is done only after sensing, i.e. after receiving a message from the server that contains sensor information. The first step is to translate the perceived message into usable objects within the perception. Next, the synchro is informed about the type of message that arrived and if a new server-cycle has begun. The synchro decides, if an action

[1] Dainamite is implemented in Java, because it was more common to participating students, accepting the loss of performance compared to C/C++

should be computed. The way it does is explained in 3. After this, the perceived information is written into the world-model. If acting, the tactic is called, which selects the appropriate high-level action (state) by comparing the available and possible ones using planning and/or estimated value functions. This mechanism is described in 4. Finally, the selected state is executed by calling the responible skill inside the action component, which maps a high-level action (state) to motions (dash, turn, kick, etc.). These are stored inside the synchro, which will send them before the cycle ends. The synchronization, the action selection and the skills are described in the following subsections in more detail.

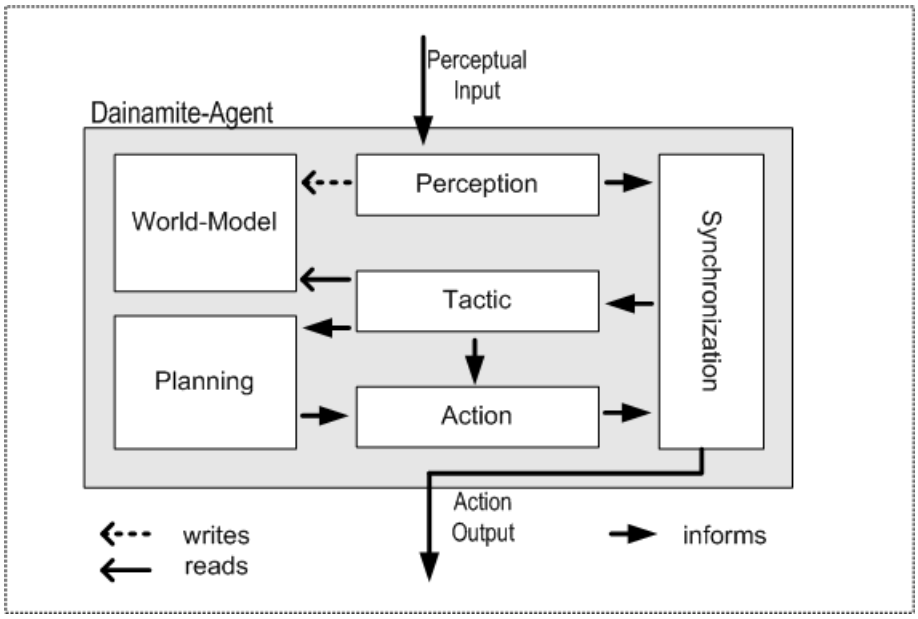

Fig. 1. Interaction of the agents internal components

3 Synchronization

As mentioned above, this component is responsible for deciding when to act. To compute actions based on the most actual world-model, it is desired to react in the majority of cases upon visual information (VI). To achieve this, the agents view mode is changed at least every two cycles, so that a VI arrives in each. The synchro tracks the arrival times of the VI and also estimates the real cycle lengths using the sensed body-state information (SBI). The decision on which message to react depends on the expected arrival time of a VI. If this is very close to the

beginning of the next cycle, an action is computed after SBI. When the next VI is received timely, the previously computed action is overwritten by a new one based on latest visual information, if not, the prior action is kept and sent before the cycle is over. In the other cases, when the VI is expected to arrive sooner, an action is only computed on VI, ignoring the SBI. The advantage of this approach is that computing actions is in most cases done only when neccessary, saving cpu effort. Another one is, that if the agent knows how much time is remaining, he can reason about the amount of alternatives to explore in his action selection and planning functions.

4 Action Selection (Tactic) and Planning

The most crucial element of an agent is his action selection, sometimes referred to as tactic, because this component decides how he behaves. Therefore a *Dainamite* agent consists of two action selection layers. The first one maps a current world-model to higher level actions (e.g. to dribbling, passing, blocking an opponent, etc.). The second maps those higher level actions to motions, which are the commands accepted by the soccer server (e.g. dash, turn, etc.). The former layer is very similar to a state machine, hence the higher level actions are called states, which all share an interface, that is used to select the best one. This is as follows:

- pre-condition
- post-condition
- success-probability
- success-benefit
- execute

The *pre-* and *post-conditions* are used as filters, removing all states from further consideration if those tests fail. The *success-probability* gives an estimated value for successful execution of the state [2], the *success-benefit* an estimated value for how good the effect is, whereas the maximum value should be the scoring of a goal. The product of both values leads to an assessment, which is used to compare states with each other. The *execute* method is the mapping to motions. Each agent has now a list of states available, from which he can select one in every cycle. The amount and types of the states depends on the role of the agent. For instance the goalie usually do not need a *shoot goal* but a *catch ball* state. State selection using this approach is very simple, following this basic algorithm.

Let S be the states available for agent a. Then the state to select is given by

$$S_{valid} = \{s \in S | pre(s) \wedge post(s)\} \tag{1}$$

$$s_{opt} = \arg \max_{s \in S_{valid}} (successBenefit(s) * successProbability(s)) \tag{2}$$

[2] Values are reaching from 0 to 1.

In (1) the states are filtered to get all those that are executable. In (2) they are assessed and the best one is selected next. This is done every time an action should be computed, as mention in Sect. 3. The difficulty in this approach is mainly to provide the probability- and benefit-values. In the current implementation, these values are estimates based on some relevant factors, such as how close am I to the opponents goal, how many opponents are near to me, etc. A more sophisticated approach would be to learn these values, whereas reinforcement learning seems to be the natural choice for the benefit, and the creation of some kind of statistic for the probabilty-values, e.g. as proposed in [2].
Another difficulty is to foresee the outcomes of the states, which is closely related to the problem of estimating the benefit-values. Especially when considering actions whose effects will eventually become true in more than a few cycles, or when more than one possible option should be observed. These problems occur for instance considering passing or intercepting the ball. For those cases, planning is done, whereas domain knowledge is used as much as possible to reduce the examined action space. Ball interception is planned by simulating all agents running to the next expected ball-position. Passing is planned by considering a large number of possible passes (differing in direction and power of the kick) and examining the possible ball interception-points of teammates and opponents. Whenever planning is done, there is no need to use the second layer of action selection, because the motion is already computed as part of the resulting plan. However, replanning is done in every cycle, because newer and more accurate information will improve the computed action or even change the agents state.

5 Skills

The skills of the *Dainamite* agents are very similar to those explained in [1], hence a detailed description is omitted. The basic approach is to construct higher level actions such as dribbling, passing or scoring a goal out of a set of lower level actions like move to a position, turn the body into a direction, and so on. But there are also some improvements, especially for the secondary action *turn neck*. This action is responsible for visual information an agent will receive next, hence the decision where to look affects the accuracy of the agents world-model notably. The task is not only to see as many objects as possible, but also to see the right objects depending on the current situation. Therefore our team uses *Situation Based Perception*, a method that determines the new look-direction depending on the actual situation. The classification of the current situation is already done by the action selection mechanism, where a state was selected. This state influences the *turn neck* action by defining weights for the visual objects in the following manner:

- ball-weight, opp-goalie weight
- weight-3m, weight-10m, weight-25m, weight-40m, far-away weight
- kickable-team, kickable-opp, team-weight, opp weight

Each weight is a positive real number, which stands for the relative importance of the considered objects. The algorithm selects a new angle to look to by maximizing the overall importance of the seeable objects, whereas it considers a discrete amount of view directions and compares them depending on the view width and the time that the resulting observable area and the containing objects were not seen lately. For most applications this approach leads to good results, gaining a good overview about the high weighted objects. One drawback however is, that in certain cases the agent should not only be interested in something (i.e. in having a high weight on it), but he should observe it without exception. For these cases, a skill exists, which turns the neck directly to a desired object.

6 Assessment of the Architecture

The *Dainamite* agent architecture presented here has some properties that seem to be favourable, at least from the point of view of the programmers. One of them is the easy way of configuring the behaviours of an agent, which can simply be done by adding or removing states. This enables the developer to test different agent configurations without changing the source code, what appears to be a good prerequisite for using genetic programming, which was already tested for example in [3]. The state concept also provides means for other learning techniques, such as reinforcement learning, because value functions are already used and could easily be replaced. Another strength of the architecture is the good synchronization with the server, which worked well even if the cycles variied in length due to cpu load on the server.
However, improvements can be made in many ways. The agents described here are in most situations short-sighted, meaning that they do not look very far into the future. Hence the extended usage of planning seem to be a reasonable approach in facing this problem. Finally, the performance of the team depends highly on the individual skills of each agent. From results of tests against other good teams follows that *Dainamite's* capabilities regarding skills can be enhanced, either by learning or by hard coding.

References

1. de Boer, R., Kok, J.: The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team. (2002)
2. Stone, P.: Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer. MIT Press (2000)
3. Luke, S.: Genetic programming produced competitive soccer softbot teams for robocup97. In Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., Riolo, R., eds.: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann (1998) 214–222