The Dainamite 2008 Team Description

Holger Endert[1], Thomas Karbe[2],
Jens Krahmann[2], Frank Trollmann[1], and Nicolas Kuhnen[2]

[1] DAI-Labor, TU Berlin
Faculty of Electrical Engineering
and Computer Science
{holger.endert|frank.trollmann}@dai-labor.de
[2] Technische Universität Berlin
Faculty of Electrical Engineering
and Computer Science
{karbe|jenskra|nkuhnen}@cs.tu-berlin.de

Abstract. This paper gives an overview of the structure of the dainamite agent, which was redesigned and implemented from scratch in order to overcome weak points of the framework used in the last year. This includes especially the modules tactic and worldmodel, but also incorporates a generic decision support module based on effective possible option analysis. Furthermore we investigated the usage of the online environment Second Life for community-based soccer match analysis and presentation, which is presented as well.

1 Introduction

The dainamite framework [1–3], which was used in the last two robocup championships, proved to be well suited for building reactive and competitive agents for the 2D simulation league. After the release of our base code [3], we noticed a few other teams, developers and researchers, who used our framework as a starting point.

However, based on our experiences we detected some weak points. First of all, the tactic was not capable of modelling longterm, multi-agent based activities. Instead, each agent individually optimized his own decision, which was recalculated every cycle. Thus, longterm planning involving different roles is quite hard to model [4]. Another weak point was the lack of a unified method of coping with the movement and ball handling of an agent. This led to an implementation with few reusable structures and methods and therefore was difficult to maintain. Similar observations were made for the worldmodel. Finally our synchronisation method was based strictly on a sense-think-act cycle, following an easy pattern allowing to act on visual information. Thus, our agents wasted a lot of time, which could be used otherwise for tactical computations.

[3] See www.dainamite.de and http://groups.google.com/group/dainamite
[4] Though we had some mechanisms supporting teamwork, like dynamic role assignment

Inspired by different approaches from other teams (e.g. [4, 5]), we redesigned and implemented our framework from scratch, taking over only those parts which proved to work well. In detail, we strictly organised the framework into different modules, which are presented in Section 2. The worldmodel was designed with an associative approach, outlined in Section 3. An improved tactic is presented in Section 4. Next, we outline our work that deals with visualising robocup matches in a virtual world (Second Life), which should facilitate the accessibility of simulated soccer to larger communities all over the world. Finally we give our conclusion and the perspectives of future work.

2 The Agent Architecture

By the term *agent architecture* we refer to the building blocks of an agent and how these are organised within the agent. From a very general point of view, this structure looks very similar in most teams, and thus we decided to first identify the dependencies and connections of the components on an abstract level. This lead to a very basic architecture, in which different components can be replaced by specific implementations, as long as they implement the given interfaces. To preserve as much flexibilty as possible, we used the Spring-Framework [5] to configure the agents runtime structure. It allows to link dependencies in XML instead of linking them directly in the code, so that replacement of components is a simple matter of configuration. In fact, each component has its own configuration, and thus may be extended as well. Futhermore we can reuse parts of the structure for building the player agents, the coach and a trainer. In that respect, an agent in our understanding is just a container, which allows to aggregate a set of required components.

An overview of the existing modules is given in Figure 1. They are presented as boxes and are connected by arrows indicating a dependency. Each module except the *DomainModel* and the *Agent* corresponds to a (set of) components, which are aggregated in order to specify the internal behaviour of an agent. The *DomainModel* is a collection of common classes (e.g. the representation for player, ball, flags, time, etc.), and may also be called our *ontology.* Therefore it is required by all other modules (except the client), but has no component state. The *Agent* is the aggregation of all components. The *Client* is responsible for interacting with the environment (i.e. with the simulator), and therefore has no dependencies. The other components are connected via the *Synchro*, which controls the internal behaviour of the agent.

On top of this architecture, we implemented an internal control cycle for the player agents. Although the protocol version 12 made synchronisation much easier, we introduced a method based on three dependent threads. An overview is given in Figure 2 as state-chart. In the upper part there is the client, which continuously receives data and writes it into a buffer. Sending lies in the responsibility of the *Synchro*, which lies in the middle of the diagram. If the send-time

[5] See http://www.springframework.org/

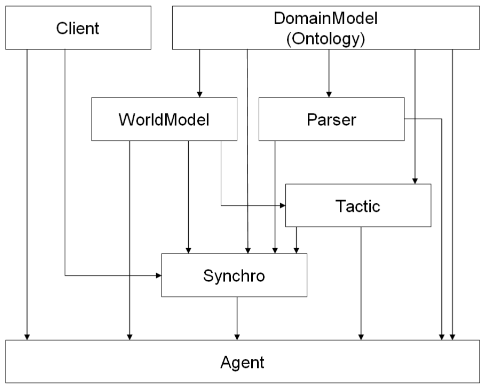

Fig. 1. Dependencies between the modules of the dainamite agent architecture

is reached (left branch), the concurrently running tactic is interrupted, the best action is retrieved and sent to the server. Thereafter the worldmodel is updated (based on predictions) and the tactic is resumed. If data is received by the client, the worldmodel is informed. The tactic runs iteratively, so that in each iteration, a new option is evaluated and stored, if it was better than every previously calculated one.

The next three sections provide some details on the components of an agent.

3 Worldmodel

We introduced two additional notions to overcome the weak points of the former worldmodel. First, we follow an associative approach, in which the object of interest is retrieved from the worldmodel by a key (called *memo*). For instance, if the own position of a player is requested, we would call the method *getPlayer(SELF)*, where the parameter is the *memo*. Accordingly, we may refer to the same object by providing the memo *TEAMMATE_7* (if the number of the player is 7), leading to very flexible access. Another advantage is, that the interface stays very slim. Internally we use hashtables for accessing the data, which is encapsulated into so called *facts* - the second concept introduced.

In our understanding, a fact represents the agents knowledge of some data (and not only the data itself). Facts can have different properties and may be organised in different layers. For instance, they may refer to sensor-data (parser output), which is located in the lowest layer, or they may refer to derived and predicted data, which is located on a higher level. In general, the level is determined by explicit dependencies. Sensor-data has no dependent facts, the ball depends on visual or time-information and the interception point needs the current positions and velocities of all mobile objects (which themselves depend on sensor-data, and so on). Furthermore, facts are responsible for themselves for being up to date. Therefore they contain timestamps and can evaluate, if dependent facts are up to date as well. This leads to an implicit tree (cycles are not

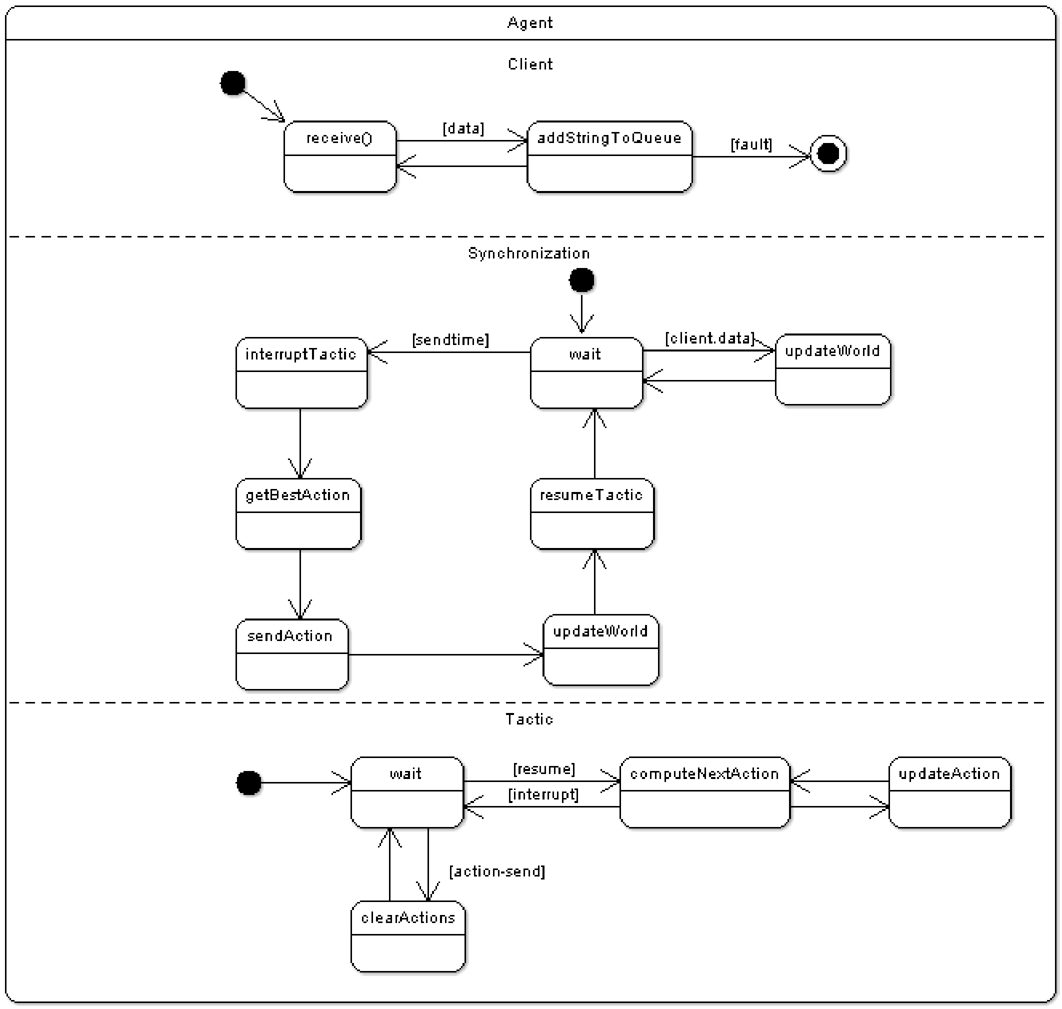

Fig. 2. The control threads of an agent (client, synchro and tactic) and their interaction.

permitted) of facts, which recalculates parts of its content each time a request is made and newer data is available. Although the tree-operations produce some computational overhead, its self-organising structure leads to good performance, if not every fact is requested in every cycle (e.g. if the ball is far away, the goalie does not care for the positions of all players).

Again, this approach is very generic. We can use the same structure for the player agents and the coach by assigning slightly different facts to the set of memos. Furthermore we can exchange parts of the worldmodel by exchanging the corresponding facts, as long as all needed memos are supported. Finally reusability is enforced, because every fact needs to be modelled explicitly w.r.t. its meaning and its dependencies.

4 Tactic

The new Dainamite action-selection mechanism is responsible for calculating the best action based on the available worldmodel. As stated in Section 2, it runs as a concurrent thread in order to achieve a trigger-independent mechanism. Since new sensor data arrives periodically, the tactic must take this into account. Whenever new information is available (indicated by a flag), the tactic decides when to integrate it [6]. This makes it possible to finish the current calculation first, if required.

Generally, the action-selection is realised with a decision tree, which incorporates the tactic of each role as a subtree. Thus we can model the individual behaviour of each teammate, are able to define cooperative activity, can reason about the goals and actions of teammates and provide the means for dynamic role assignment. Further this structure has the advantage of being very modular and convenient to extend.

In order to build such trees the nodes are assigned distinct meanings. Those which are not leafs (called *Nodes*), are used to define the tactic by means of conditioned options. The leafs (called *State Nodes*) then decide what happens. They contain so called *States*, which are responsible for calculating specific actions. By traversing through the tree to a *State Node*, the best action can be found. An example tree is given in Figure 3. There two paths from root *Node* to *State Nodes* are shown. The *States* are drawn inside the containing *State Nodes*. Within the *Nodes* playmodes, roles of the agents and situations are assessed. The leafs are all *State Nodes* that result from the previously assessed factors.

The three main constituents, the *Nodes*, *State Nodes* and *States* are described in more detail in the following sections.

4.1 Nodes

Nodes are the main structures of the tactic tree. Each *Node* has a *precondition*. The precondition is a filter that removes the *Node* from further consideration if it proves to be false.

[6] It operates on copies of the data contained in the worldmodel.

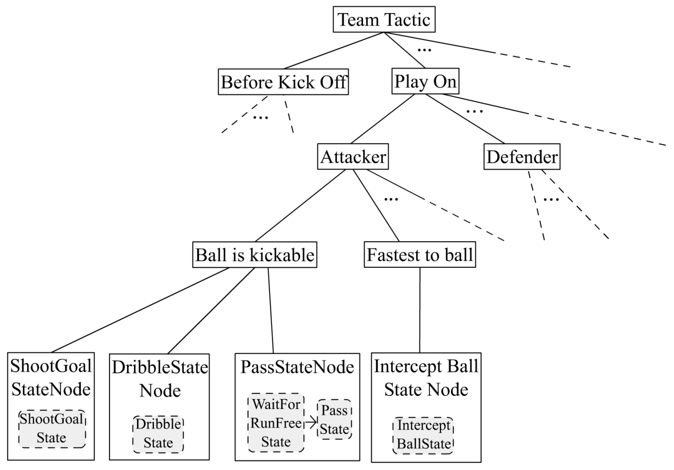

Fig. 3. An excerpt from a possible Decision Tree

When traversing the decision tree in Figure 3 the selection of child *Nodes* happens in three stages. First we filter the nodes by their preconditon. If the child nodes are *State Nodes* they are then assessed based on the evaluation functions of the contained *States*. Finally the *Node* with fulfilled precondition and highest evaluation function (if applicable) is selected.

4.2 State Nodes

The *State Nodes* represent the leafs of the tree and therefore the decision for a general behaviour model. A *State Node* can contain a list of *States*. These are executed in sequential order, providing planlike capabilities.

Additionally a *State Node* provides functions to determine wether the current *State* can be kept in the next cycle or has to be discarded. This way a *State* can stay active for more than one cycle and the tactic doesn't have to start over calculating everything from scratch. If the *State* can't be kept, the precondition of the successor (if one exists) decides wether to use it or rather start over calculating the best action at the root of the tree.

Thus the *State Nodes* represent a more complex behaviour than a single *State* because they can be used longer than just one cycle and specify a sequence of activities. They can be used to plan actions, giving the plan the flexibility to abort once the situation has changed. This way even joint activity can be modelled by using a sequence of *States* for every role and basing the decision for changing to the next node on the *states* the other actors are in.

In Figure 3 for example the *State Node* "Pass State Node" contains the two *States* "Wait For Run Free State" and "Pass State" . This means, in order to pass the ball to a player we first have to wait for him to be free and then pass him the ball. The Player will stay in the *State* "Wait For Run Free State" until the other player has a good position or it has to be discarded. This could happen if an opponent comes too close.

4.3 States

A *State* represents the basic behaviour for a given situation. Each *State* has a precondition which determines wether it can be used in a given situation or not. Additionaly a *State* provides functions to evaluate the utility and thereby compare several applicable *States* to find the most promising one. The task of a *State* is to calculate the best possible action for the situation it represents. It calculates the main action, and based on that all secondary actions that should be sent to the server.

For example the *State* "Shoot Goal State" in Figure 3 generates the actual Kick-Action for shooting the goal. Based on that action the secondary actions are then calculated. The turn neck action for example can be set to turn the head towards the direction the ball will be kicked.

5 Second Life Visualisation

The good visualisation of complex systems (like RoboCup) is an essential requirement for understanding its behaviour. Furthermore the ability to share knowledge and opinions with experts of the domain may help to get insight into occuring problems. Since experts may be rare and spread all over the world, we investigated the idea of visualising RoboCup games [7] in Second Life. Second Life is a virtual online world, where the huge community has the possibility to create its own world. As an avatar, which is the inworld representation, a member can walk or fly through the virtual world or communicate to another avatar. Our intention is to achieve a visualisation similar to a common monitor but directly in the 3D world. So it is possible to walk through the simulation. Residents of Second Life may like to join the visualisation and can get an easy access to the world of RoboCup simulation this way.

5.1 Architecture

Second Life offers good options for building visual representations, but has a rather limited programming interface, the Linden Script Language (LSL). LSL is an event driven script language, which can be used to control the behaviour of inworld objects. The language offers over 300 library functions. There are functions which enable the script to communicate with the user or other scripts in the same or other objects. It is also possible to request data via http. However, a lot of restrictions exist in the scripts, e.g. they can only use 16kb of data. So complex application have to be build of multiple scripts.

[7] Currently restricted to 2D-Simulation

Given these facts, each player is represented through autonomous Second Life objects, which obtain the data for the cycles by http calls. The data contains information about the position, rotation, colour of the player and colour of the kickable of the player. In order control and synchronise the visualisation, a controller object is needed. This has an interface to the avatar and may be used to create the simulation objects (player, ball, field, etc.) and start, pause and end the simulation.

In the backend, data is accessed not directly from the server, but from a conventional monitor (SoccerScope). SoccerScope is a java-based monitor for the 2d simulation. It can connect to the running server and playback logfiles. It saves all data in a internal worldmodel, which gives an easy access to the data. The http based protocol is integrated into SoccerScope.

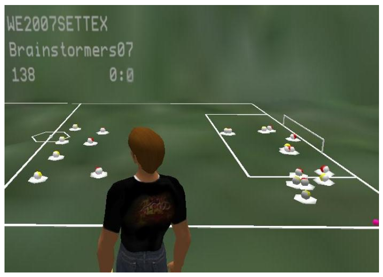

Fig. 4. Watch the 2007 wm final

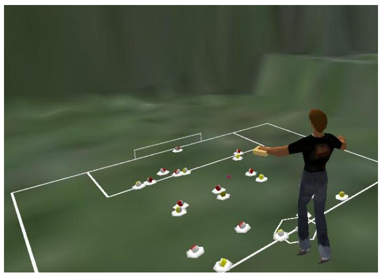

Fig. 5. fly over the game

References

1. Endert, H.: The dainamite 2006 team description (2006)
2. Endert, H., Wetzker, R., Karbe, T., Heßler, A., Brossmann, F.: The dainamite 2007 team description (2007)
3. Endert, H., Wetzker, R., Karbe, T., Heßler, A., Brossmann, F., Büttner, P.: The dainamite agent framework. Technical report, DAI-Labor, Technische Universität Berlin, Germany (2006)
4. de Boer, R., Kok, J.: The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team. (2002)
5. Berger, R.: Die doppelpass-architektur verhaltenssteuerung autonomer agenten in dynamischen umgebungen. Diploma thesis, Institut für Informatik, Humboldt Universität zu Berlin (2006)