Inductive Verification and Validation of the KULRoT RoboCup Team

Kurt Driessens, Nico Jacobs,
Nathalie Cossement, Patrick Monsieurs, Luc De Raedt

Dept. of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
http://www.cs.kuleuven.ac.be/~nico/robocup

Abstract. As in many multi-agent applications, most RoboCup agents are complex systems, hard to construct and hard to check if they behave as intended. We present a technique to verify multi-agent systems based on inductive reasoning. Induction allows to derive general rules from specific examples (e.g. the inputs and outputs of software systems). Using inductive logic programming, partial *declarative* specifications of the software can be induced. These rules can be readily interpreted by the designers or users of the software, and can in turn result in changes to the software. The approach outlined was used to test the KULRoT RoboCup simulator team, which is briefly described.

1 Introduction

The RoboCup simulator offers a rich environment for the development and comparison of multi-agent systems (MAS). These systems are often very complex. An agent must cope with incomplete and partially incorrect data about the world it acts in, the goal of the agent isn't formally defined ('play good football') and the system operates in real-time as a consequence of which the timing-issue is very important. Because of all this, the design and programming of a single agent is a difficult task; verifying that the agent behaves as described in the design is even harder as is the verification and validation (V&V) of MAS (see section 3).

In the area of verification and validation of knowledge based systems there have been attempts to adjust the V&V systems towards the problem of verifying and validating MAS (see for instance [13]). However most of these systems assume that the agents act upon a knowledge base and that one can specify the intended behavior of the knowledge base, which is not the case for many MAS.

The V&V method we propose is based on induction. Rather than starting from the specification and testing whether it is consistent with an implementation, inductive reasoning methods start from an implementation, or more precisely, from examples of the behaviour of the implementation, and produce a (partial) specification. Provided that the specification is declarative, it can be interpreted by the human expert. This machine generated specification is likely to give the expert new insights into the behaviour of the system he wants to verify. If the induced behaviour is conform with the expert's wishes, this will

M. Asada and H. Kitano (Eds.): RoboCup-98, LNAI 1604, pp. 193-206, 1999.
© Springer-Verlag Heidelberg Berlin 1999

(partly) validate the system. Otherwise, if the expert is not satisfied with the induced specification, he or she may want to modify the knowledge based system and repeat the verification or validation process.

This paper addresses the use of inductive reasoning for verification and validation of MAS. The employed techniques originate from the domain of inductive logic programming, as these methods produce declarative specifications in the form of logic programs. The sketched techniques are tested in the RoboCup domain. This paper is organized as follows: in section 2, we introduce inductive learning through inductive logic programming; in section 3 we show how this technique can be used in verification. In section 4 we show how we used our technique to verify our agents, after which we discuss related work and conclude. The agents are described in appendix A. For the remainder of this article we assume the reader is familiar with Prolog [18].

2 Inductive Logic Programming

Inductive logic programming [15] lies at the intersection of machine learning and computational logic. It combines inductive machine learning with the representations of computational logic. Computational logic (a subset of first order logic) is a more powerful representation language than the classical attribute-value representation typically used in machine learning. This representational power is necessary for verification and validation of knowledge based systems, because such knowledge based systems are in turn written in expressive programming languages or expert system shells. Another advantage of inductive logic programming is that it enables the use of background knowledge (in the form of Prolog programs) in the induction process.

An ILP system takes as input examples and background knowledge and produces hypotheses as output. There are two common used ILP settings which differ in the representation of these data: learning from entailment ([7] compares different settings) and learning from interpretation [10]. In this paper we will use the second setting. In learning from interpretations, an example or observation can be viewed as a small relational database, consisting of a number of facts that describe the specific properties of the example. In the rest of the paper, we will refer to such an example as a model. Such a model may contain multiple facts about multiple relations. This contrasts with the attribute value representations where an example always corresponds to a single tuple for a single relation.

The background knowledge takes the form of a Prolog program. Using this Prolog program, it is possible to derive additional properties (through the use of Prolog queries) about the examples. If for instance we are working in a domain where family-data is processed, possible background knowledge would be:

parent(X,Y) ← mother(X,Y). parent(X,Y) ← father(X,Y).
grandmother(X,Y) ← mother(X,Z), parent(Z,Y).

There are also two forms of induction considered here: predictive and descriptive induction. Predictive induction starts from a set of classified examples

and a background theory, and the aim is to induce a theory that will classify all the examples in the appropriate class. On the other hand, descriptive induction starts from a set of unclassified examples, and aims at finding a set of regularities that hold for the examples. In this paper, we will use the TILDE system [1] for predictive induction, and the CLAUDIEN system [9] for descriptive induction.

TILDE induces logical decision trees from classified examples and background theory. Consider for example this background knowledge:

```
replaceable(gear).  replaceable(wheel).  replaceable(chain).
not_replaceable(engine).  not_replaceable(control_unit).
```

and a number of models describing worn parts and the resulting action (in total 15 models were used):

```
begin(model(1)).    begin(model(2)).    begin(model(3)).   ...
sendback.           fix.                keep.
worn(gear).         worn(gear).         end(model(3)).
worn(engine).       end(model(2)).
end(model(1)).
```

TILDE will return this classification tree:

```
worn(A) ?
+--yes: not_replaceable(A) ?
|       +--yes: sendback
|       +--no:  fix
+--no:  keep
```

CLAUDIEN induces clausal regularities from examples and background theory. E.g. consider the single example consisting of the following facts and empty background theory:

```
human(an).   human(paul).   female(an).   male(paul).
```

The induced theory CLAUDIEN returns is:

$$\text{human}(X) \leftarrow \text{female}(X). \qquad \text{human}(X) \leftarrow \text{male}(X).$$
$$\text{false} \leftarrow \text{male}(X) \wedge \text{female}(X). \qquad \text{male}(X) \vee \text{female}(X) \leftarrow \text{human}(X).$$

Notice that this very simple example shows the power of inductive reasoning. From a set of specific facts, a general theory containing variables is induced. It is not the case that the induced theory deductively follows from the given examples. Details of the TILDE and CLAUDIEN system can be found in [1, 9].

3 ILP for Verification and Validation

Given an inductive logic programming system, one can now verify or validate a knowledge based or multi-agent system as follows. One starts constructing examples (and possibly background knowledge) of the behaviour of the system

to be verified. E.g. in a knowledge based system for diagnosis, one could start by generating examples of the inputs (symptoms) and outputs (diagnosis) of the system. Alternatively, in a multi-agent system one could take a snapshot of the environment at various points in time. These snapshots could then be checked individually and also the relation between the state an agent is in and the action it takes could be investigated.

Once examples and background knowledge are available one must then formulate verification or validation as a predictive or descriptive inductive task. E.g. in the multi-agent system, if the aim is to verify the properties of the states of the overall system, this can be formulated as a descriptive learning task. One then starts from examples and induces their properties. On the other hand, if the aim is to learn the relation among the states and the actions of the agent, a predictive approach can be taken.

After the formulation of the problem, it is time to run the inductive logic programming engines. The results of the induction process can then be interpreted by the human verifiers or validators. If the results are in agreement with the wishes of the human experts, the knowledge based or multi-agent system can be considered (partly) verified or validated. Otherwise, the human expert will get insight into the situations where his expectations differ from the actual behaviour of the system. In such cases, revision is necessary. Revision may be carried out manually or it could also be carried out automatically using knowledge revision systems (see e.g. Craw's KRUST system [5], or De Raedt's Clint [6]). After revision, the validation and verification process can be repeated until the human expert is satisfied with the results of the induction engines.

4 Experiments in RoboCup

In this section we describe some verification experiments. For this, a preliminary version of the agents described in appendix A was used. The most important differences are that the agents in the experiments did not yet used predictions about the future position of ball and players and that the decision-network used (figure 2) was much simpler.

4.1 Modeling the Information

The first tests were run to study the behavior of a single agent. We supplied the agents with the possibility to log their actions and the momentary state of the world as perceived by them. This way we were able to study the behavior of the agents starting from their beliefs about the world. Because all the agents of one team were identical except for their location on the playing field, the log files were joined to form the knowledge base used in the experiments. A sample description of one state from the log-files looks as follows :

```
begin(model(e647)).
   player(my,1,-43.91466,5.173167,3352).
   player(my,2,-30.020395,7.7821097,3352).
```

```
    ...
    player(other,10,14.235199,15.192206,2748).
    player(other,11,0.0,0.0,0).
    ball(-33.730022,10.014952,3352).
    mynumber(5).
    bucket(1).
    rctime(3352).
    moveto(-33.730022,10.014952).
    actiontime(3352).
end(model(e647)).
```

The different predicates have the following meaning :

$player(T, N, X, Y, C)$ the agent has last seen the player with number N from team T at location (X, Y) at time C.

$ball(X, Y, C)$ the agent has last seen the ball at location (X, Y) at time C.

$mynumber(N)$ this state was written by the agent with number N. It thus corresponds to the observation of agent N.

$bucket(N)$ the bucket used for bringing the agent back to its home position. The bucket-value was lowered every input/output cycle and forced the agent to its home-location and reset when it reached zero.

$rctime(C)$ the time the state was written.

$actiontime(C)$ the time the action listed was executed.

$moveto(X, Y), shoottogoal, passto(X, Y), turn(X), none$ the agent's action.

The $rctime(C)$ predicate was used to judge the age of the information in the model as well as to be able to decide how recent the action mentioned in the model is. This was done by comparing the $rctime(C)$ with $actiontime(C)$. Logging was done at regular time intervals instead of each time an action was performed, so we could not only look at why an agent does something, but also why an agent sometimes does nothing. The time-units used were the simulation-steps from the soccer-server.

Some of the actions that were used while logging were already higher level actions than the ones that can be sent to the soccer-server. However these actions, such as *shoottogoal* for example, were trivial to implement.

To make the results of the tests easier to interpret an even higher level of abstraction was introduced in the background knowledge used during the experiments. Actions that are known to have a special meaning were renamed. For instance a soccer-player that was looking for the ball always used the $turn(85)$ command, so this command was renamed to *search_ball*. An other example of information defined in the background knowledge is the following rule :

```
action(movetoball):- validtime, moveto(X1,Y1), ball(X2,Y2),
                      distance(X1,Y1,X2,Y2,Dist), Dist =< 5 .
```

in which the $moveto(X, Y)$ command was merged with other information in the model to give it more meaning. For instance, $moveto(-33.730022, 10.014952)$ and $ball(-33.730022, 10.014952, 3352)$ in the model shown above, would be merged into *movetoball* by this rule. Often a little deviation was permitted to take the

dynamics and noise of the environment into account. The actions used to classify the behavior of the agent were : *search_ball*, *watch_ball*, *moveto*, *movetoball*, *moveback*, *shoottogoal*, *passto*, *passtobuddy* and *none*. Some of these actions were not used in the implementation of the agent but were included anyway for verification purposes. For instance, although — according to specifications — an agent should always "move to the ball" or "move back", the possible classification *moveto* was included in the experiments anyway, to be able to detect inconsistencies in the agent's behavior..

Other background knowledge included the playing areas of the soccer-agents and other high level predicates such as *ball_near_othergoal*, *ball_in_penaltyarea*, *haveball* etc. Again, not all of these concepts were used when implementing the agent. This illustrates the power of using background knowledge. Using background knowledge, it is possible for the verifier to focus on high-level features instead of low-level ones.

4.2 Verifying Single Agents

The first tests were performed with TILDE, which allowed the behavior of the agent to be classified by the different actions of the agent. The knowledge base used was the union of the eleven log-files of the agents of an entire team. The agents used in the team all had the same behavior, except for the area on the field. The area the agent acted in depended on the number of the agent and also was specified in the used background knowledge. The resulting knowledge base consisted of about 17000 models (14 Megabyte), collected during one test-game of ten minutes. The first run of TILDE resulted in the following decision tree

```
seeball ?
+--yes: ball_in_my_area ?
|       +--yes: haveball ?
|       |       +--yes: ball_near_othergoal ?
|       |       |       +--yes: action(shoottogoal) [15 / 15]
|       |       |       +--no:  action(passtobuddy) [122 / 124]
|       |       +--no:  action(movetoball) [1007 / 1015]
|       +--no:  bucket_was_empty ?
|               +--yes: action(moveback) [342 / 347]
|               +--no:  action(watch_ball) [2541 / 3460]
+--no:  action(search_ball) [7770 / 7771]
```

Only about 12000 models were classified. We did not include the action *none* as a classification possibility because although a lot of models corresponded to this action, it's selection was a result of the delay in processing the input information instead of depending on the state of the agent's world. Most of the classifications made by TILDE were very accurate for the domain. However, the prediction of the *action*(*watch_ball*) only reached an accuracy of 73,4%.

To get a better view on the behavior of the agent in the given circumstances CLAUDIEN was used to describe the behavior of the agent in case "*seeball*, *not*(*ball_in_my_area*), *not*(*bucket_was_empty*)." CLAUDIEN found two rules that

describe these circumstances. The first rule was the one TILDE used to predict the *watch_ball* action.

```
action(watch_ball) if  not(action(none)), seeball,
                       not(ball_in_my_area), not(bucket_was_empty).
```

CLAUDIEN discovered the rule had an accuracy of 73%. The other rule that was found by CLAUDIEN was the following :

```
action(moveback) if  not(action(none)), seeball ,
                     not(ball_in_my_area), not(bucket_was_empty).
```

which reached an accuracy of 26%. It states that the agent would move back to its home location at times it was not supposed to. Being forced to go back to its home-location every time the bucket was emptied, this behavior was a result of the bucket getting empty while the player was involved in the game and therefore not paying immediate attention to the contents of the bucket.

To gain more consistency in the agents behavior, the bucket mechanism was removed and replaced by a new behavior where the agent would move back when it noticed that it was to far from its home location. The new behavior, after being logged and used in a TILDE-run resulted in the following tree :

```
seeball ?
+--yes: ball_in_my_area ?
|       +--yes: haveball ?
|       |       +--yes: ball_near_othergoal ?
|       |       |       +--yes: action(shoottogoal) [48 / 48]
|       |       |       +--no:  action(passtobuddy) [85 / 85]
|       |       +--no:  action(movetoball) [796 / 810]
|       +--no:  at_place ?
|               +--yes: action(watch_ball) [3826 / 3840]
|               +--no:  action(moveback) [384 / 394]
+--no:  action(search_ball) [7180 / 7318]
```

in which the *action*(*watch_ball*) was predicted with an accuracy of 99,6 %. The increase in consistency in the behavior in the agent, improved its performance in the RoboCup environment. Because the agent only moved back to its home-location when necessary it could spend more time tracking the movement of the ball and fellow agents and therefore react to changing circumstances faster.

4.3 Verifying Multiple Agents

In agent applications it is often important that not only all agents individually work properly, the agents also have to cooperate correctly. One important point in this is to check if the beliefs of the different agents more or less match. In the case of our RoboCup agents we want to know for instance if there is much difference between the position where player A sees player B and the position where player B thinks it is[1]. So we used CLAUDIEN to find out how often agents have different believes about there positions, and how much their beliefs differ.

[1] it is impossible to know what the real position of a player is, so we can only compare the different believes the agents have.

To do these tests, we transformed the datafile so that one model contains the believes of multiple agents at the same moment in time. CLAUDIEN found multiple rules like the one below:

```
Dist < 2 if mynumber(A,Nr), vplayer(A,my,Nr,X1,Y1), vplayer(B,my,Nr,X2,Y2),
            mynumber(B,Nr2), vplayer(B,my,Nr2,X3,Y3), not(A=B),
            distance(X1,Y1,X2,Y2,Dist),distance(X2,Y2,X3,Y3,Dist2),Dist2<10.
```

This rule, which has an accuracy of 78% states that if two players are less then 10 units apart, the difference in the believes of the position of one of those two players is less then 2 units. From all the rules we could conclude useful information, for instance, we found out that our agents can best estimate a team mate's position from distance 10. All rules found were 'acceptable' rules (e.g. for distances larger than 10, the error is positively correlated with the distance between the players), so we can conclude from the observed behavior that the beliefs of the different agents do not differ much.

5 Related work

This work builds upon earlier ideas on combining verification and validation with inductive logic programming [8]. It is also related to other approaches applying machine learning with validation and verification. This includes the work of Susan Craw on her KRUST system for knowledge refinement [5], the work by Bergadano et al. and the work by De Raedt et.al. [11]. The approach taken in KRUST is complementary to ours. Rather than starting from examples of the actual behaviour of the system, KRUST starts from examples of the desired behaviour of the system. Whenever the two behaviours do not match, KRUST will automatically revise the knowledge based system. It is clear that the KRUST approach could also be applied within our methodology, at the point where the human discovers inconsistencies between the two behaviours. If the human then specifies examples of the intended behaviour, KRUST might help revising the original knowledge base. The approach of Bergadano et. al. and De Raedt et. al. using inductive machine learning to automatically and systematically generate a test set of examples that can be used for verification or validation. Finally, our work is also related to the work by William Cohen [3] on recovering software specifications from examples of the input-output behaviour of the program.

6 Conclusions

We sketched a novel approach to verification and validation, based on inductive reasoning rather than deduction. We reported a number of experiments in the domain of MAS (RoboCup) which prove the concept of the approach.

Further work on this topic could involve applying the verification and validation technique also to other multi-agent systems (such as e.g. DESIRE [2]), and also to extend the inductive method to other representations. For instance, it seems very well possible to apply inductive techniques in order to automatically construct decision tables starting from the knowledge base. Such decision tables

are already popular in V&V, but they are typically made by the human expert (in collaboration with the machine), see e.g. [19].

Acknowledgements: The authors wish to thank Hendrik Blockeel and Luc Dehaspe for their help with the TILDE and CLAUDIEN system. Nico Jacobs is financed by a specialisation grant of the Flemish Institute for the promotion of scientific and technological research in the industry (IWT). Luc De Raedt is supported by the Fund for scientific research, Flanders. This work is supported by the European Community Esprit project no. 20237 (ILP 2).

A Team Description

A.1 Introduction

In this appendix we describe the KULRoT team for the RoboCup '98 simulator league [12]. It is the result of some preliminary experiments in the domain, and the main emphasis is on detecting problems related to building multi agent systems for the RoboCup task and using machine learning techniques to overcome these problems. To simplify the building of a RoboCup team the complete team consists of identical players[2] which only differ in their field position.

The agents are implemented in Java for different reasons. The main reason is that Java is platform independent, an important aspect for code being simultaneously developed on different operating systems. Moreover using Java it's easy to use multi-threading. The use of the UDP protocol [17] is also embedded in Java.

This description is structured as follows: in section A.2 a general overview of the structure of the soccer agent is presented, in section A.3 we discuss the beliefs held by an agent. The acting of the agent is split up in low level skills (section A.4) and high level skills (section A.5). Finally in section A.6 we describe some timing problems and how we tackled these.

A.2 General overview of the soccer agent

The general structure of each player is presented in figure 1. It consists of five main parts:

- **Communicator**: this module acts as an intermediate layer between the real agent and the soccer server; it mainly manages the sockets for communication.
- **Sensors**: this thread parses the incoming information about what the agent sees and hears. At the moment the full input string gets parsed and the information stored in the world model.
- **World model**: the information received from the sensors is used to update the world model. This model is however an active model in the sense that it is able to predict the (approximate) position of objects in the future using the formulae the server uses to calculate the trajectory of objects. This is explained in more detail in section A.3.

[2] except for the goalie.

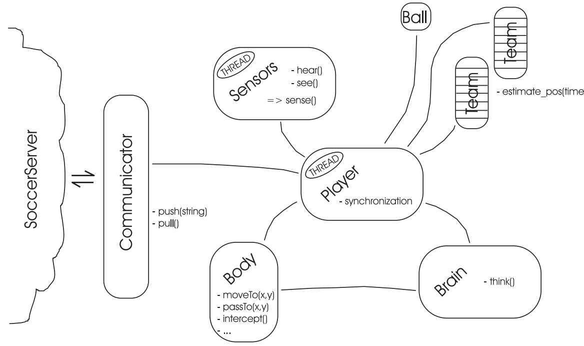

Fig. 1. General overview of the soccer agent

- **Body**: this module contains the low level skills of the agent. These are skills like turning with a ball, moving to a certain position or intercepting a ball. It uses the world model for this. See section A.4 for a description of these low level skills.
- **Brain**: this is the module in which the real decision making occurs. This module decides based on the world model which actions need to be executed and translates these to low level commands, which are then sent to the body module. This is explained in more detail in section A.5.

The player class only starts up the other modules and is used to let the other modules communicate with each other.

A.3 Beliefs and World Model

The information about the world stored within the agent is modeled using absolute coordinates. The see-messages arriving from the soccer server are parsed by the Sensors class and transformed to absolute coordinates.

Although the modeling in absolute coordinates requires extra work during the processing of the sensor information, it limits the updates required when changing the position of the agent and enables easier reasoning about future locations of moving objects.

The algorithm available in the libsclient library [16] was translated and used to calculate the absolute coordinates and facing direction of the agent. This information is then used to triangulate the absolute coordinates of the other objects present in the see-string.

The memory of the player consists of field objects that represent the twenty-two players and the ball. They hold the information needed to calculate their position at a given time in the near future, i.e. the coordinates they were last seen by the agent, the speed and direction they were traveling in at that moment and the simulator time they were last seen at. This, together with the mathematical formulas that represent the course of the object that are used within the soccer server allows the field objects to estimate their future positions. This information will become more and more inaccurate when the object hasn't been seen a long

time or when looking far into the future. Also the estimations will be wrong when the object deviates from its course at which it was last seen.

Because every object on the field has its own representation as an object of the field object class, it is not easy to use information about players of which the number or even the team is not visible. As a consequence, this information is not used. Seen objects are only considered by the player when all identifying information is available. This of course limits the long range view of the agent and may be changed in the future. For the same reason, the low quality setting for the sensor information which is available in the soccer server is not used by the agent at the moment, because it supplies the sensory information without the identifying information necessary to use it.

A.4 Low Level Skills

The low level skills of the agent are represented by, and implemented in the Body class. Most of the low level skills implemented in the agent are concerned with transforming actions in the used coordinate system to actions which can be performed by the agent through the soccer-server.

Such actions are $moveTo(x, y)$, $passTo(x, y)$ or $shootToGoal()$. Starting from the agents own position and direction, either as observed or predicted, the necessary actions are calculated and performed. During these calculations, consideration is also paid to the fact whether the player stands between the ball and the target. For timing issues discussed later, the time the action will end is also calculated and returned as a result of the action.

Based on these actions, it was possible to supply the agent with a bit more complicated actions on a low level. Such actions as $markPlayer(number, team)$, $dribble()$, $moveAndCheck(x, y)$ or $turnWithBall()$ which are independent of the current field- or team-situation were implemented at this level were checks about own and target position could be evaluated half way during the action.

Successfully intercepting the ball requires a combination of three things the agent must performs. First it must estimate the time it will take to reach the ball. Then the player must estimate the position of the ball at that time. To complete the intercept, it must move to that position. The last two actions are already discussed above. The tricky part is estimating the best future time-instance to intercept the ball at.

Different strategies were tried to accomplish this. The first implementation looked a fixed number of simulator steps ahead and moved the agent to the estimated position of the ball at that time. Because neither the distance to the ball, nor its speed, nor the direction it is traveling in are considered this way, the method was not very accurate nor successful.

The second strategy calculated the intercept-time by starting from an underestimation and increasing this value by one simulator step until a time-instance was found at which the soccer-agent could reach the estimated position of the ball at that same time. This method was more successful but had computation requirements too large to be useful in the real time environment of RoboCup.

The solution used by the KULRoT team was obtained by generating a large set of examples of intercept-times together with seven relevant values : distance

to the ball, the relative view-angle of the ball, the relative travel direction of the ball, the travel direction of the player, the speed of the ball, the speed of the player and the player's current effort. This set of examples was then used to generate a decision tree with TILDE [1] which predicted the intercept time based on the values given. The success rate of the intercept using this decision tree was comparable with the one that calculated the correct time — 59% vs. 63%, the reason for the failing of the intercept with the correct calculations being the error in the estimations of the ball speed and location — but the calculation time was much lower.

Preliminary tests using neural networks resulted in a lower success rate. Other strategies (e.g. using perpendicular intercept trajectories) were tested as well, but with bad results. More information can be found in [4].

A.5 High Level Skills

In the previous section we discussed some actions that the agent can perform. In this section we discuss how we decide which action to perform at a certain moment in time. The basic decision structure is based on a network structure depicted in figure 2, which can also be seen as a tree with the rightmost node as the root.

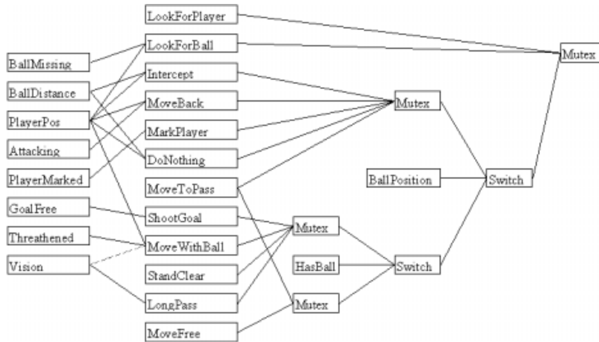

Fig. 2. Selecting high level actions

The network consists of two types of nodes: value-nodes and conditional nodes. A basic value node returns a value based on the world model. All leaves of the tree are basic nodes. For instance the basic node GoalFree returns a higher value if there are less 'obstacles' between itself and the opponents' goal. Besides basic value nodes there are also combined value nodes. These nodes combine values from other nodes into a new value. For instance the combined value node LookForBall combines the values of BallMissing and PlayerPosition: if the value for PlayerPosition is low (this means the player is close to its standard field position) and the value for BallMissing is high (which means it has been a long time since the player last saw the ball) the value for LookForBall will be high.

Results of value-nodes can also be the inputs for conditional nodes, of which there are two types. A mutex node takes multiple values as input and returns the highest value as output. A switch node takes three value-nodes as input and returns either the first or the third value, based on whether the value of the second node is below some threshold.

Everytime the agent can act, the tree is evaluated. The value nodes in the second column are each related to one high level action. The root node will return the value of one of those value nodes, and the high level action corresponding to this value node will be executed. This high level action (for instance looking for the ball) has to be translated to low level actions (e.g. multiple turn commands) which are then sent to the body of the player to be executed.

An important question is how to weigh all the values in combined value nodes. For instance: is the negative influence of PlayerPosition larger than the positive influence of BallMissing on the combined value node LookForBall? To solve this problem all input links to combined value nodes are given a weight (either positive or negative), which makes this a parameter optimization problem. This is solved by implementing a genetic algorithm, with the weight vector as elements of the population, and the result of a RoboCup simulator game with a team of players using these weights in their network as the fitness function. More information on the subject of high level skills can be found in [14].

A.6 Timing the Actions

One of the most difficult tasks for the agent was making sure it was working with up-to-date information. Because all the actions of the agent depend on the correct estimation of the agents own position and other field objects position, it is important to have the agents the world information correct and up-to-date at the time a new action is chosen.

Because of the non continuous way the see-information is provided to the agent, a possibility of delay between the actions performed and the related visual information exists. This delay can originate from visual information which originated during the agent's action and was not interpreted as such.

To take care of this problem, the agent now estimates the end-time of every action it performs. Then, if it wants to be sure the information about the world is accurate and up to date, it can wait for visual information that originated in the soccer server after the end-time of its last action.

To make this possible the current simulator step was kept in a clock within the agent which updated itself every 100 milliseconds and synchronized itself with the time given in see- or sensebody-messages. This enabled the agent to know the precise time an action was chosen and the time it was started by sending it to the soccer server and as a consequence to calculate the time the action should be performed.

This solution was preferred above the more obvious one of comparing the position of the agent or other field objects — such as the ball — to the target position of the action. The choice was based on the fact that actions cannot be guaranteed to succeed so an agent moving to a field location (x, y) could wait

indefinitely for its action to end if, for instance, another player was standing in its way.

References

1. H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–85. Springer-Verlag, 1997.
2. F. Brazier, B. Dunin-Keplicz, N. R. Jennings, and J. Treur. Desire: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6:67–94, 1997. Special Issue on Formal Methods in Cooperative, Information Systems.
3. W. Cohen. Recovering Software Specifications with ILP. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 142–148, 1994.
4. N. Cossement. Robocup: developing low level skills. Master's thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998.
5. S. Craw and D. Sleeman. Knowledge-based refinement of knowledge based systems. Technical Report 95/2, The Robert Gordon University, Aberdeen, UK, 1995.
6. L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.
7. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
8. L. De Raedt. Using ILP for verification, validation and testing of knowledge based systems, 1997. invited talk at EUROVAV 1997.
9. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.
10. L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.
11. L. De Raedt, G. Sablon, and M. Bruynooghe. Using interactive concept learning for knowledge-base validation and verification. In *Validation, Verification and Test of Knowledge-based Systems*, pages 177–190, 1991.
12. H. Kitano, M. Veloso, H. Matsubara, M. Tambe, S. Coradeschi, I. Noda, P. Stone, E. Osawa, and M. Asada. The robocup synthetic agent challenge 97. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 24–29. Morgan Kaufmann, 1997.
13. N. Lamb and A. Peerce. Verification of multi-agent knowledge-based systems. In *Proceedings of the ECAI-96 Workshop on Validation, Verification and Refinement of Knowledge Based Systems*, 1996.
14. P. Monsieurs. Developing high level skills for robocup. Master's thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998.
15. S. Muggleton and C. D. Page. A learnability model for universal representations. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, pages 139–160, Sankt Augustin, Germany, 1994. GMD.
16. I. Noda. Libsclient (for c language). URL: http://ci.etl.go.jp/˜noda/soccer/client/index.html.
17. J. Postel. RFC 768: User datagram protocol, 1980.
18. Leon Sterling and Ehud Shapiro. *The art of Prolog.* The MIT Press, 1986.
19. J. Vanthienen, C. Mues, and C. Wets. Inter-tabular verification in an interactive environment. In *Proceedings of the '97 European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV-97)*, pages 155–165, 1997.