VERTEILTE SYSTEME

UNIKASSEL VERSITÄT

Efficient Decision Making in ALICA

University of Kassel

Project Report from

Stephan Opfer

At

Distributed Systems

Reviewer: Prof. Dr. Kurt Geihs

Supervisor: Dipl.-Inf. Hendrik Skubch

Dipl.-Inf. Michael Wagner

May 28, 2009

Abstract

In a highly dynamic environment like a football match for a team of autonomous robots, efficient decision making is one of the best conditions to win. This project work focuses on the choice between alternative plans and the task allocation of plans. The team behaviour is specified with ALICA - A Language for Interactive Cooperative Agents. The approach is based on the A* Search Algorithm combined with utility functions to evaluate alternative task allocations and plans. The result is a decision making, which takes less than $1ms$ for all strategic decisions of one agent. The project work was successfully deployed in the Carpe Noctem Software Architecture and used in the RoboCup domain.

Contents

1 Introduction **4**

2 Foundations **5**

2.1 Agent - Environment Relationship 5
2.2 A* Search 5
2.3 ALICA 6

3 Motivation and Problem Definition **10**

3.1 Cooperative Teams of Autonomous Robots 10
3.2 RoboCup and Carpe Noctem 11
3.3 Decisions in Football Matches 11
3.4 Task Definition 12

4 Implementation **16**

4.1 Integration of the PlanSelector 16
4.2 A* based Task Allocation 19
4.3 Handling of Hierarchical Structures 20
4.4 Evaluation of Allocations 23

5 Evaluation and Future Work **26**

5.1 Evaluation 26
5.2 Future Work 28

6 Summary **29**

Bibliography **30**

1 Introduction

The *RoboCup* initiative provides highly dynamic environment for different kinds of autonomous multi agent teams. One of these teams is the Carpe Noctem team of the University of Kassel. In 2008/09 the most significant improvement for this team, thus far has been its new behaviour engine. This engine is the first implementation of ALICA - A Language for Interactive Cooperative Agents. With this language it is possible to specify plans that describe how a team of robots should act together. In the course of the adoption of ALICA a new decision making mechanism was necessary. In the RoboCup domain it is very important to have an efficient decision making mechanism, because of its highly dynamic environment. This project work complies with this requirements by using an adopted A* search algorithm and utility functions to evaluate the alternative possibilities. This project work focuses on the choice between alternative plans and the task allocation of plans.

In Chapter 2, the general foundations are described. It contains the A* search algorithm, ALICA and an abstract description of the agent-environment relationship. Chapter 3 motivates the topic of this project work and gives an exact problem definition. Chapter 4 contains the implementation. The implementation comprises the integration into the new behaviour engine, the decision making mechanism itself and a description of the utility functions with some examples. Chapter 5 is spitted up into two parts. The first part gives an extract of the comprehensive evaluation of this project work. The second part lists up ideas for future work, which occurred during the implementation and evaluation of this project work. Chapter 6 completes this elaboration with a concluding summary.

2 Foundations

This chapter explains the fundamental topics that are the prerequisites this project work is based on. At first the relationship between an agent and its environment is described. Afterwards the A*-Algorithm is explained, because it is the core algorithm of this project work. This chapter ends with a short description of the domain specific language *ALICA*.

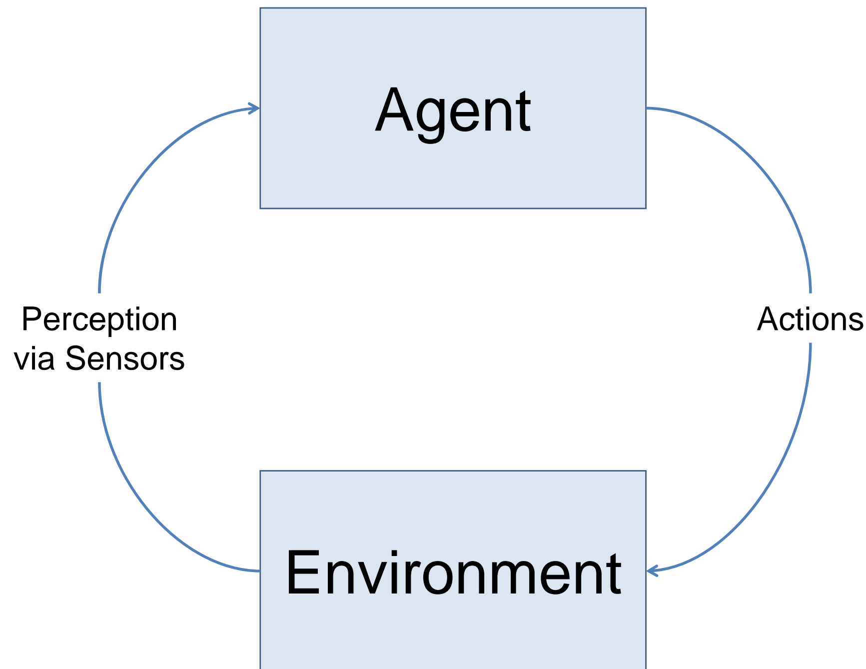
2.1 Agent - Environment Relationship

There is no unique definition for the term agent, because an agent has depending on the domain or environment very different properties. In this section a description of the agents, this project work is implemented for, is given. An agent is a robot with several sensors and actors, which allows it to interact with its environment as illustrated in Figure 2.1. Assuming an agent to be rational, means that it *"acts so as to achieve the best outcome or, when there is uncertainty the best expected outcome."* [1] Together the agents build an autonomous multi-agent team, which communicates important parts of their belief bases.

2.2 A* Search

A Search* is a best-first search algorithm first described in 1968 by Hart et al. [4]. It evaluates a node on the basis of the costs it takes to get to this node from the start and also on the estimated costs it takes to get from this node to the goal node. The costs to get from the current node to the goal node are estimated by a heuristic function. Under the condition that the heuristic function is optimistic and consistent the *A* Search Algorithm* is admissible and computationally optimal [3]. Optimistic means that the heuristic function never overestimates the real costs to get from a node to the goal node. A heuristic function is consistent, if it is true for every node that the estimated costs to get from this node to

[1] Russell and Norvig [7], Page 4, Chapter 1



/2/2

Figure 2.1: Agent and Environment Interaction

the goal node are less or equal than the costs to get from this node to a successor node plus the estimated costs to get from the successor node to the goal node. Algorithm 1 and 2 depict the *A* Search* [7]. According to the explained conditions the first node in the fringe is always the node with the least estimated costs on the path to the goal node.

2.3 ALICA

ALICA is an abbreviation of the self-explanatory name "*A* *L*anguage (for) *I*nteractive *C*ooperative *A*gents". The language is a domain specific language for multi-agent teams and was developed in 2008 by Skubch et al. [9]. The syntax and semantic of the language is shortly described as follows:

A plantype is a set of plans. A plan contains a set of states, tasks and parameters. To every plan belongs a utility function as well as pre-, runtime-, and post-conditions. Util-

Algorithm 1 TREE-SEARCH($problem, fringe$) **returns** a solution, or failure

1: $fringe \leftarrow$ INSERT(MAKE-NODE(INITIAL-STATE[$problem$]), $fringe$)
2: **loop**
3:   **if** EMPTY?($fringe$) **then**
4:     **return** $failure$
5:   **end if**
6:   $node \leftarrow$REMOVE-FIRST($fringe$)
7:   **if** GOAL-TEST[$problem$] applied to STATE [$node$] succeeds **then**
8:     **return** SOLUTION($node$)
9:   **end if**
10:   $fringe \leftarrow$INSERT-ALL(EXPAND($node, problem$), $fringe$)
11: **end loop**

Algorithm 2 EXPAND($node, problem$) **returns** a set of nodes

1: $successors \leftarrow$ the empty set
2: **for all** $< action, result >$ **in** SUCCESSOR-FN[$problem$](STATE[$node$]) **do**
3:   $s \leftarrow$ a new NODE
4:   STATE[s] $\leftarrow result$
5:   PARENT-NODE[s] $\leftarrow node$
6:   ACTION[s] $\leftarrow action$
7:   PATH-COST[s] $\leftarrow$ PATH-COST[$node$]+STEP-COST($node, action, s$)
8:   DEPTH[s] $\leftarrow$ DEPTH[$node$] $+ 1$
9:   add s to $successors$
10: **end for**
11: **return** $successors$

ity functions estimate how useful their corresponding plans are, according to the current situation. The preconditions must hold to execute a plan. The runtime conditions must hold during the plan execution. The post condition holds after executing a plan. In a plan, conditional transitions leading from one state to another and span a directed state graph. Each task of a plan determines another initial state. The requirements and the function of every state graph are described by the corresponding task. One task can be contained by different plans and therefore attaches the same information and description to different state graphs. A state contains a set of Behaviours and plantypes. The described elements of ALICA build a hierarchical structured tree as shown in Figure 2.2. A *Behaviour* is a leaf of the tree (labelled with $\perp$) and describes an agents basic behaviour. A Behaviour is atomic and does not perform interaction with other agents. All other elements of this tree branch as shown. The meaning of the dashed edge is that it is always possible to reach another subplan by passing a path of a task, a state and a plantype.

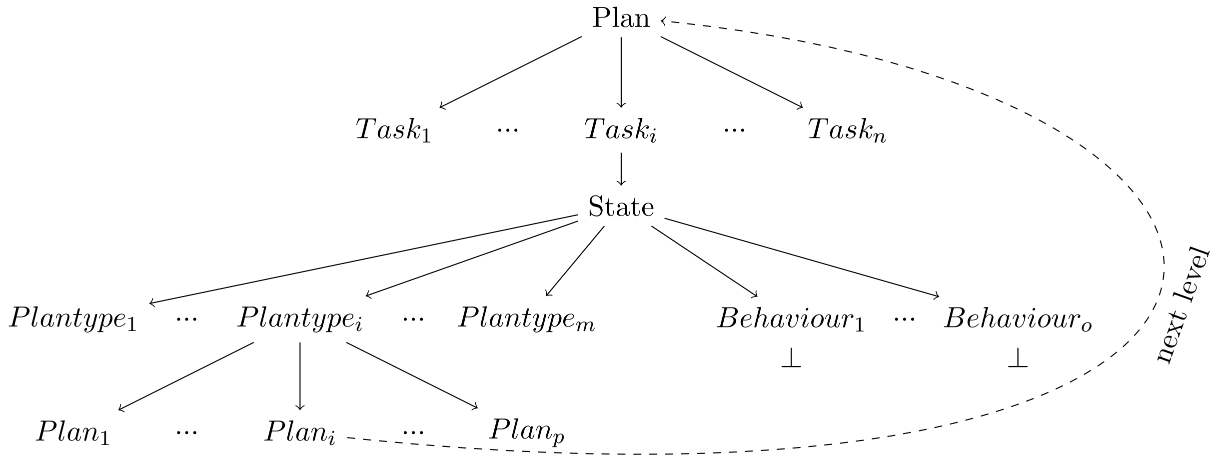
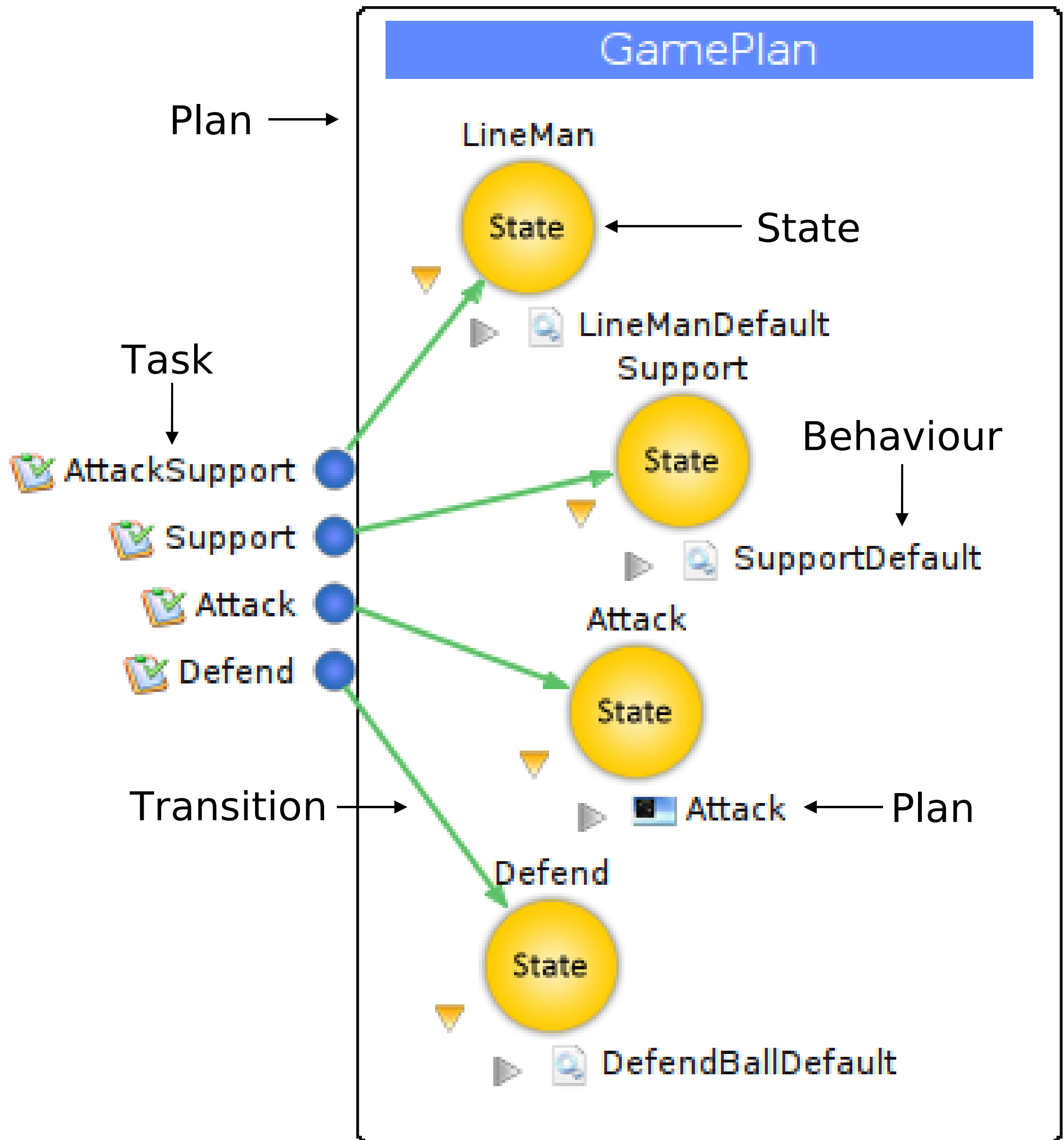


Figure 2.2: Hierarchical Structure of ALICA

An explicit interaction with other agents is done by synchronisations. A *Synchronisation* connects transitions of one plan and enforces that the transitions can only be passed simultaneously. To pass the transitions simultaneously at least one agent has to be in the original state of each transition and the conditions of each transition have to be complied.

Another element of ALICA is the role. A role consists of characteristics that describe the capabilities of an agent. With a role assigned to an agent it is possible to decide if it is capable to perform a task. Figure 2.3 shows screenshots of *PlanDesigner*. The PlanDesigner is a tool to describe ALICA conform agent behaviour. It was the bachelor thesis of Andreas Scharf in 2008 [8].



*2/5*2/5 *3/5*3/

Figure 2.3: PlanDesigner Screenshots

3 Motivation and Problem Definition

This chapter starts with a short introduction into domains where teams of autonomous robots can be used. Section 3.2 introduces the RoboCup domain and the RoboCup team *Carpe Noctem*, because this work is integrated in the *Carpe Noctem Software Architecture* and evaluated in the RoboCup domain. Section 3.3 describes some typical decisions in a football match and points out which of this decisions are made by this project work. Finally this chapter ends with a clear definition of the problem solved by this project work.

3.1 Cooperative Teams of Autonomous Robots

Cooperative teams of autonomous robots are used in various domains. For example, Rus et al. [6] use a team of autonomous robots to move furniture. The robots must cooperate to manipulate heavy furniture. Rus et al. [6] claim that the robots achieve their task without any global control and explicit communication. Another domain is rescue robots that are able to find human beings in collapsed buildings after an earthquake [5]. Small rescue robots crawl into the collapsed buildings while an autonomous quadcopter collects thermal images to locate hidden fires. With the knowledge about the location of the different fires the small robots could form teams and split up over the area to quench the different sources of fire to save the human beings. Yet another domain is autonomous football playing robots. Two teams of autonomous robots play against each other a football match with rules similar to the FIFA rules [1]. These are three different domains, but all have similar requirements posed on the autonomous robots. The robots have to cooperate, they have to make correct decisions and they have to adapt their behaviour according to the changing environment to achieve their goals.

[1] http://www.fifa.com/worldfootball/lawsofthegame.html last accessed 2/14/2009

3.2 RoboCup and Carpe Noctem

> RoboCup™is an international research and education initiative. Its goal is to foster artificial intelligence and robotics research by providing a standard problem where a wide range of technologies can be examined and integrated. [...] The main focus of the RoboCup activities is competitive football. The games are important opportunities for researchers to exchange technical information. [...][2]

With these words the RoboCup initiative describes itself, its ambitions and claims that a football match provides problems for a wide range of research areas. This project work is principally involved in the RoboCup domain, too. *Carpe Noctem*[3] is the RoboCup team of the University of Kassel, which successfully participated in several RoboCup competitions. Those competitions are separated into different leagues like Humanoid League, Rescue League, Standard Platform League, Small Size League, Simulation League, and Middle Size League. *Carpe Noctem* is a Middle Size League team. To conceive an idea of the dimensions of this league and because this project work is evaluated in this league, some differences to a normal football match are described here. Since 2009, each team consists of five robots, one goalkeeper and 4 field players. Each robot is about 80 cm high and fits into a square of 50cm x 50cm. The field is 12m wide and 18m long. One match lasts two equal periods of 15 minutes. Depending on the team's hardware, a robot can drive up to 5m/s and can shoot the ball over a distance of 10m. Based on that velocity and shooting range it is obvious that there are ever-changing situations during a RoboCup football match, forcing the robots to quickly adapt their behaviour to the situation.

3.3 Decisions in Football Matches

Bisanz and Gerisch [2] claim at pages 184 and 185 in Section "Wahrnehmungs- und Entscheidungsschnelligkeit", that it is typical for high class football players to quickly make correct decisions with just one look at the current game situation. Furthermore, they claim that the skill to make fast decisions gives a bigger room for manoeuvre and leads to better performance. This also applies to robotic football matches. In a talk at the 1. RoboCup Workshop Kassel 2008 - „Role assignment and behavior control" the RoboCup team from

[2]http://robocup.org/Intro.htm last accessed 2/14/2009

[3]http://carpenoctem.das-lab.net/ last accessed 01/20/2009

the University of Osnabrück[4] presented that their robots have a reaction delay of about 150ms[5], which is already a good performance. These 150ms include among others the decision making. Hence, an efficient decision making directly improves the reactivity of the robot. For instance if the decision making would aproximately take 80ms, a ball with a velocity of 10m/s would travel a distance of 80cm during this time. In the Middle Size League the penalty point is just 3m away from the goal. With a 50cm wide goalkeeper the ball passed a third of the distance to the goalkeeper before the goalkeeper has decided to which side it should move to save the ball. With additional delay due to image processing and motor reaction it is impossible to save a penalty shoot. Similar situations arise during an ongoing match. With decision making which takes about 80ms the line of defence is not built quickly enough to stop the opponent or in general it is impossible to react fast enough to prevent any action of the opponent. Decisions such as hinted at in the last sentences are part of the task definition of this project work. From an abstract point of view the question is which strategy is the best for the current situation and which part of the strategy is allocated to which robot. A pass as a simple example includes two tasks - one to pass the ball and one to receive the pass. This simple example has already a lot of possible errors. There could be two or more robots which want to receive the pass and by moving into position, they handicap each other. Another option is that no robot wants to receive the pass. Yet another possibility is that making the decision to pass or to take the task of receiving the pass takes too much time and the enemy have enough time to intercept the pass. So it is important that every task is allocated to the correct number of robots and it is important that the allocation is done in less than 10ms.

3.4 Task Definition

The goal of this project work is to develop a software module called *PlanSelector*. The PlanSelector has to be integrated in the new behaviour engine [10] of the *Carpe Noctem Software Architecture*. One instance of the new behaviour engine, with an integrated PlanSelector module, is running on each robot. The PlanSelector module distinguishes between the *own/ local robot* and the *other robots* of the team. There are several decisions and tasks the PlanSelector module is responsible for. The PlanSelector module has to decide which task allocation of a plan can provide the highest utility result of the plan's utility function.

[4]http://www.ni.uos.de/ last accessed 02/15/2009
[5]http://carpenoctem.das-lab.net/research/robocup-workshop last accessed 02/15/2009

This decision is made by the PlanSelector module, given a set of robots, the current situation and a given parameter substitution. Another task of the PlanSelector module is to decide which plan of a set of plans provides the highest utility result according to their best task allocation. In this case, all plans are contained by a plantype. A plantype represents a set of alternative plans. This means that the PlanSelector module makes strategic decisions choosing one of several alternative plans. Almost every plan spans a hierarchical structured tree of subplans as described in Section 2.3. Decisions as described before have to be made just for the part of the tree the own robot is allocated for. There are three parameters the PlanSelector module's decisions are based on:

Set of Robots The set of robots includes all robots, which are in the same state as the own robot.

Set of Plans The set of plans includes all plans and plantypes of the state the own robot is in.

Situation The situation is represented in the robot's belief base.

For a better understanding of the task definition an extensive example is given. A task allocation of one plan can be seen as shown in Table 3.1. A "1" in cell (i, j) means that robot r_i is allocated to task τ_j, a "0" means that it is not allocated to task τ_j. There are some constraints for a valid allocation. Only one "1" per robot (column) is allowed. Each task has a minimum and a maximum cardinality. These cardinalities define how much robots are needed minimal and how much robots can help maximal to achieve the goal of the task. The number of robots allocated to a task must satisfy these cardinalities. The allocation also has to satisfy the precondition and the runtime condition of the plan. Only a valid allocation is able to achieve the goal of the plan.

p	r_1	r_2	r_3
τ_1	1	0	0
τ_2	0	0	1
τ_3	0	1	0

Table 3.1: Task Alloction Matrix

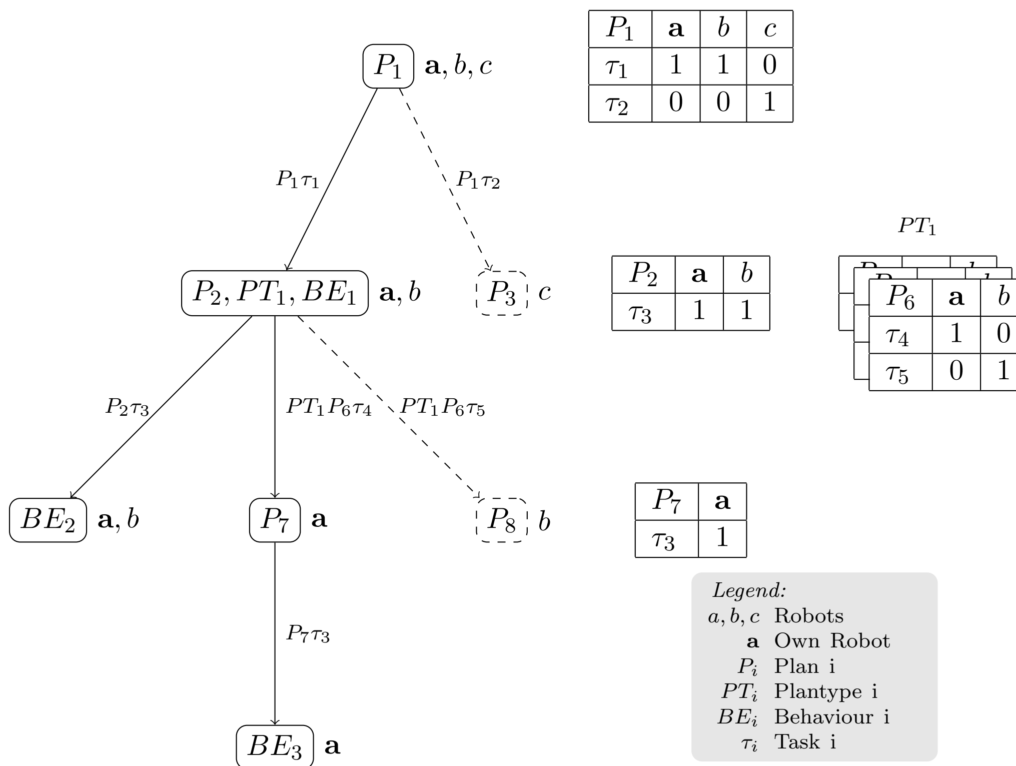
Plantypes add a new dimension to the matrix as shown in Figure 3.1. The new dimension consists of the set of plans in a plantype. According to this, a single plan can be seen as a plantype with just one plan.

p_3	r_1	r_2	r_3
τ_1	1	0	0
τ_2	0	1	1

Figure 3.1: Task Allocation Matrices for a Plantype

Figure 3.2 shows an almost complete *task allocation tree*. The nodes of this tree represent states in terms of ALICA. As explained in Section 2.3, states contain behaviours and plantypes. To simplify matters, plantypes with just one plan are represented as single plans. The robots in a state are noted on the right side of it. The edges of the tree represent the tasks which the robots are allocated for. The dashed part of the tree is the part the PlanSelector module need not calculate. The PlanSelector module just has to calculate the part its own robot (**a**) is allocated for. The allocations for the state which includes P_7 depends on the allocation of its parent state in the tree. That means if τ_3 of P_7 has a minimal cardinality of 2 robots, the PlanSelector module has to backtrack to the parent state and try to change its allocation. For example, it could allocate robot b for the same task as robot a. It can happen that the PlanSelector module has to backtrack the complete tree and try all possible combinations of allocations without success. In such a case the PlanSelector module returns a failure message.

After this chapter ended with the task definition of the project work, the following chapter deals with problem solving and gives more details about the project work.



P_1	**a**	b	c
τ_1	1	1	0
τ_2	0	0	1

P_2	**a**	b
τ_3	1	1

P_6	**a**	b
τ_4	1	0
τ_5	0	1

P_7	**a**
τ_3	1

Figure 3.2: Example of a Task Allocation Tree

4 Implementation

This chapter explains this project work in detail. At first, Section 4.1 deals with the integration in the current *Carpe Noctem Software Architecture*, especially with the integration in the new behaviour engine [10]. Section 4.2 continues with the explanation of the task allocation search. Afterwards Section 4.3 describes the handling of the hierarchical structured of plantypes including the backtracking policy. After the raw procedures are processed Section 4.4 explains the utility functions and the evaluation of task allocations.

4.1 Integration of the PlanSelector

The *Carpe Noctem Software Architecture* is pictured in Figure 4.1 [1]. The central part of it is the Base. Every other part is connected to it and the Base itself contains the *Ego World Model* and the *Shared World Model* of the robot. Together, both world models represent the belief base of a robot. Therefore, all sensing modules of the architecture fuse their information into the world models of the Base. The third part in the Base is the behaviour engine, which controls the actions of the robot. As mentioned before the PlanSelector is a submodule of the behaviour engine. For more information about the other software modules of the *Carpe Noctem Software Architecture* consult the PhD thesis of Philipp Baer [1], Section 3.4 of the diploma thesis of Stefan Triller [10], or visit the website of the *Carpe Noctem RoboCup Team*[1].

In Figure 4.2 an overview about the architecture of the behaviour engine is given. The PlanSelector module is highlighted in that figure. The PlanHandler and the PlanMonitor module are important for this project work and explained in more detail. However, the other modules are described in note form first.

Parser The *Parser* module reads the hierarchical plantype structure from XML-based files and creates the object structure accordingly to this files. This is done during the

[1] http://carpenoctem.das-lab.net/ last accessed 01/20/09

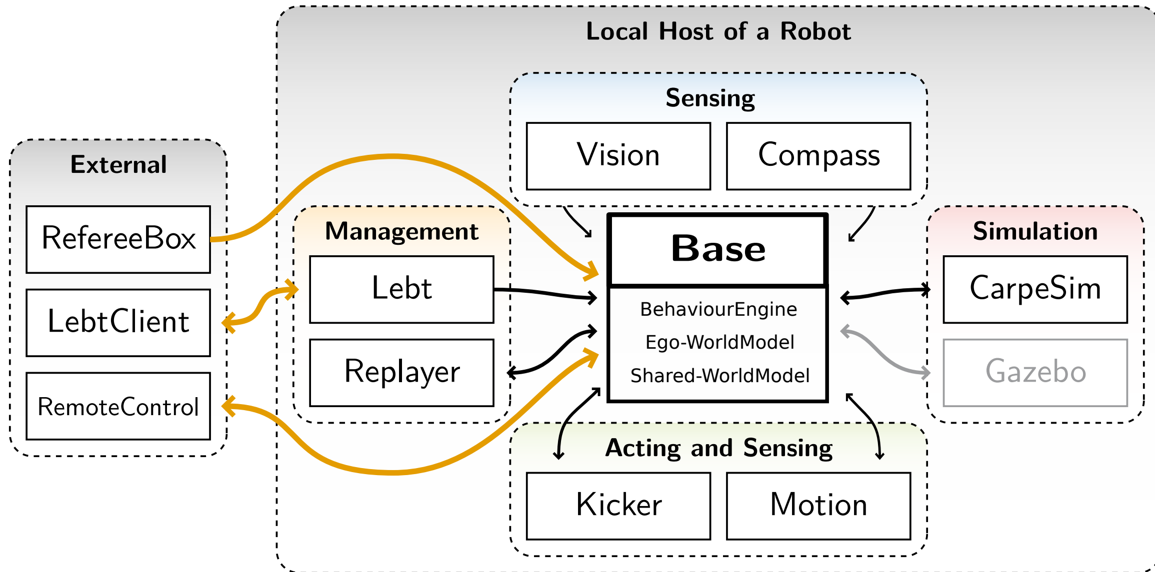


Figure 4.1: CarpeNoctem Software Architecture

initialisation in two steps. The first step is creating the objects itself and the second step is to connect the objects according to the parsed references.

Repositories There are two repositories. One stores the plan data structures and the other stores the roleset data structures. Both repositories are filled by the parser during the initialisation.

BehaviourPool The *BehaviourPool* module contains all behaviours, which can be executed. Some behaviours are executed at a specified frequency and some are triggered under special circumstances. No matter which kind of behaviour it is, the PlanHandler determines if it should be executed at the moment or not.

RoleAssignment The *RoleAssignment* maps one role to each robot, according to its abilities. The PlanSelector module asks for the role of a robot, to decide if it is capable for the different possible tasks.

ExpressionValidator The *ExpressionValidator* is able to evaluate all existing conditions. These are runtime conditions, preconditions and post conditions.

One of the more important modules for this project work is the *PlanMonitor* module. The PlanMonitor is the central module of the behaviour engine. It consists of one loop, which monitors several conditions and values: Runtime conditions of the own robot's plans, utility

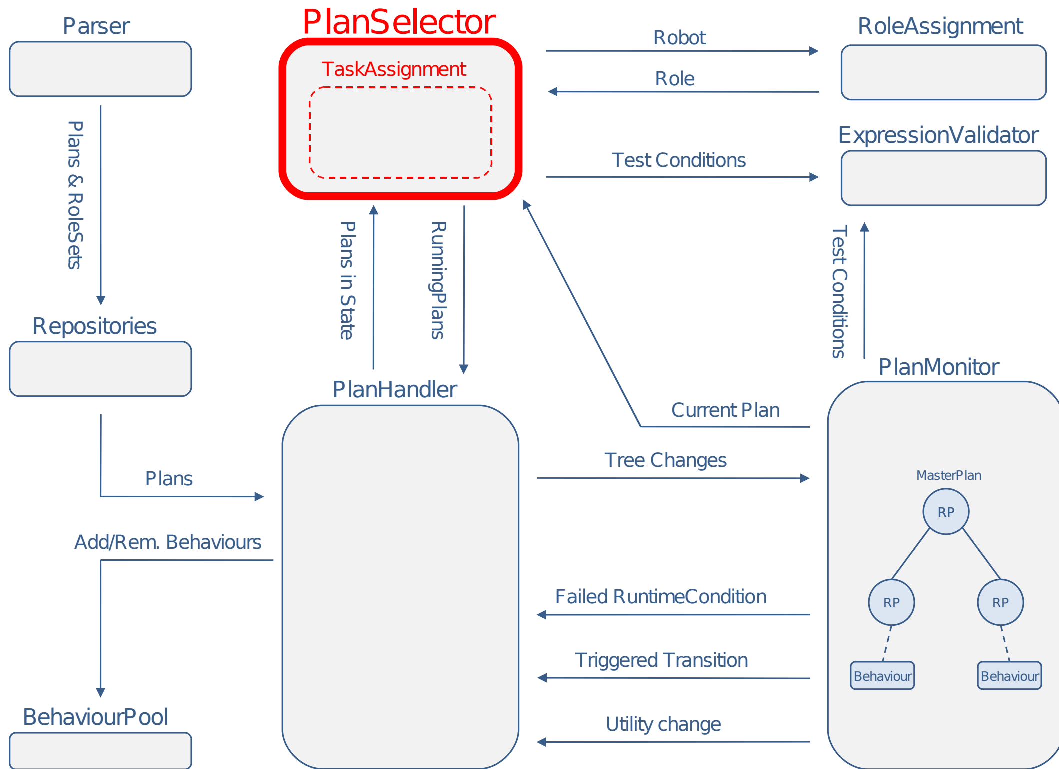


Figure 4.2: Behaviour Engine Architecture

values of task allocations of the own robot's plans and conditions of outgoing transitions of the own robot's states.

The PlanMonitor evaluates the assignments of the own robot's plans according to their utility functions. For each plan the PlanSelector is triggered to compute the best assignment for the current situation. There are three cases to distinguish. The first appears if the best assignment is already the current assignment. In this case the utility value of the plan's assignment has to be updated with the just evaluated utility value. In the second case the best assignment differs from the current, but the difference of both utility values is not bigger than the threshold of the plan. This threshold is defined according to the plan's utility function. If the threshold is exceeded a change from the current assignment to the best assignment is warranted. The third case is similar to the second, but the difference of both utilities is big enough to change the assignment. If the own robot is affected by this

change the PlanHandler module is notified about this.

The *PlanHandler* module is responsible for changes of the robot's own plan tree structure. This plan tree structure is an important part of the robots belief base and also very similar to Figure 3.2. There are different reasons to change it. For example, a transition could be triggered, a runtime condition failed or, as mentioned before, the interval between the utility values of the current and the best task allocation is big enough, so that it is advantageous to change the task allocation. For whatever reason the PlanHandler module has to change the robots own plan tree structure, it always calls the PlanSelector module. Therefore the PlanHandler passes all relevant plans, plantypes, and behaviours to the PlanSelector module and receives, if the task allocation for all levels of the hierarchy is possible, the new part of the task allocation tree. The PlanHandler module processes this part and adds or removes behaviours from the BehaviourPool, accordingly. The differences which concern the monitoring are given back to the PlanMonitor module.

4.2 A* based Task Allocation

Algorithm 3 depicts the core of the task allocation algorithm implemented by this project work. In Figure 4.2 the core is separated by a dashed rectangle inside the PlanSelector module. The core is separated from the rest of implementation to gain an ease of exchangeability for different plan structures like chains, trees or graphs. In general, Algorithm 3 is based on the A* search algorithm of Section 2.2. Algorithm 3 searches for the best task allocation of a plantype, according to the current situation and the robots suitability. The algorithm has two parameters: the plans of the plantype and the robots of the team, which are in the same state as the PlanSelector module's own robot.

The first instruction initialises the search fringe with one node per given plan. It is an ordered queue of search nodes. The fringe is ordered descending by the possible utility value of the nodes. In the while loop the first node of the fringe is removed. If the current node is a *goalnode*, it is stored. Finally, the EXPAND instruction creates every possible successor node of the current node. A successor node has one more robot assigned to a task considering its minimum and maximum cardinalities. For every task a new successor node is created, if the robot can be assigned to it.

If the while loop is left, either all possible task allocations were tried or a node passed the GOAL-TEST. A *goalnode* represents an allocation where all robots are assigned (atleast

assigned to do nothing) and the cardinality of each entrypoint have to be satisfied by the number of assigned robots. These requirements are checked by the Goal-Test. With a valid *goalnode* the corresponding allocation is constructed. Only with this allocation the plan precondition and the plan runtime conditions can be validated. If the allocation meets the conditions it is returned, otherwise the while loop is entered again to search for the next best node.

Algorithm 3 Allocation-Search($plans, robots$)

```
1:  fringe ← Insert-All(Initial-Nodes(plans, robots))
2:  repeat
3:    while ¬fringe.IsEmpty() ∧ goalnode = Null do
4:      node ←Remove-First(fringe)
5:      if Goal-Test(node) then
6:        goalnode ← node
7:      end if
8:      fringe ←Insert-All(Expand(node)) {assign next robot to all possible tasks}
9:    end while
10:   if goalnode = Null then
11:     return Null {all Allocations were tried}
12:   end if
13:   allocation ←Create-Allocation(goalnode)
14: until plan.Precondition(allocation) ∧ plan.RuntimeCondition(allocation)
15: return allocation
```

Algorithm 3 cannot handle the hierarchical plan structure in Figure 2.2. This is possible with the extended algorithms described in Section 4.3.

4.3 Handling of Hierarchical Structures

While Algorithm 3 calculates the allocation for just one plantype, the algorithms of this section calculates a complete task allocation tree. Figure 4.3 shows the task allocation tree from Figure 3.2 as it is calculated by the algorithms of this section. The tree is calculated top down and the nodes of the tree are structures called *RunningPlan*. Each RunningPlan contains an allocation for one plantype and not for a complete state as in Figure 3.2. A RunningPlan also contains an initial state. The initial state is determined by the task of a plan, which is allocated to the PlanSelector's own robot. For each plantype or behaviour in the initial state a child node is created. In case of a behaviour the allocation and initial state field is Null. If it is a plantype and the allocation field is Null, no valid allocation

exists. If the allocation field is not NULL, but the initial state field is NULL, the own robot is not allocated for any task or allocated to do nothing, respectively. The dashed parts of Figure 3.2 are omitted, because they won't be calculated by the PlanSelector of robot **a**.

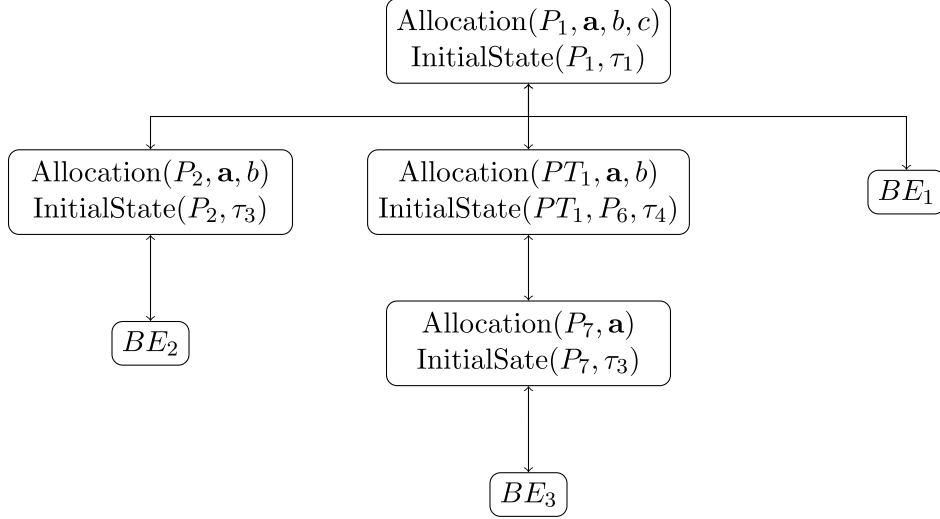


Figure 4.3: Task Allocation Tree with RunningPlans

Algorithm 4 distinguishes between plantypes and behaviours. The PlanSelector has to calculate an allocation for all behaviours and plantypes in the *abstractplans* parameter. In case of a behaviour, the PlanSelector has nothing to calculate, it is just capsuled in a *RunningPlan* and added to the result list. In case of a plantype, Algorithm 4 calls Algorithm 5 to create a RunningPlan structure with the corresponding allocation and initial state. The parameters of Algorithm 5 are all *plans* of the plantype and all given *robots*.

Algorithm 4 TYPEHANDLING(*abstractplans*, *robots*)

```
1: for all abstractplans do
2:    if TYPEOF(abstractplan) = Behaviour then
3:       RunningPlan ← abstractplan
4:    else if TYPEOF(abstractplan) = Plantype then
5:       listOfPlans.ADDALL(abstractplan.plans)
6:       RunningPlan ←CREATERUNNINGPLAN(listOfPlans, robots)
7:    end if
8:    resultList.ADD(RunningPlan)
9: end for
10: return resultList
```

A TaskAssignment object represents an allocation search for a plantype as in Algorithm 3.

According to Algorithm 3, the first instruction of Algorithm 5 adds one search node per given plan to the fringe of the TaskAssignment object. Afterwards the repeat-until loop is entered and one allocation per iteration is tried. Each iteration begins with the search for the next best valid allocation. This is capsuled in Algorithm 6, which basically contains the main part of Algorithm 3. If the allocation meets the conditions of the chosen plan, the RunningPlan structure is created with the corresponding allocation and initial state. The search continues on the next level of the task allocation tree, by calling Algorithm 4 again. Hence, Algorithm 4 and Algorithm 5 call each other to descend the task allocation tree recursively. The repeat-until loop is left, if the recursive call does not return NULL. Otherwise the repeat-until loop continues and the next allocation is tried. If all allocations are tried the next higher level of the allocation tree has to calculate another task allocation. If this chain reaction reaches the root of the task allocation tree the PlanSelector has tried every possible task allocation and has to signal to the behaviour engine that it is impossible to execute the given plantypes and behaviours at the moment.

Algorithm 5 CREATERUNNINGPLAN(*plans*, *robots*)

1: *taskAssignment* ←INIT-TASKASSIGNMENT(*plans*, *robots*)
2: **repeat**
3: *allocation* ← *taskAssignment*.GETNEXTBEST()
4: **if** *allocation* = NULL **then**
5: **return** NULL {All allocations were tried}
6: **else if** ¬*plan.conditions*(*allocation*) **then**
7: *continue*{plan conditions are violated by this allocation}
8: **end if**
9: *RP.allocation* ← *allocation*
10: *RP.initialState* ←GETINITSTATE(*allocation*, *ownRobot*)
11: *RP.children* ←TYPEHANDLING(*RP.initialState.abstractplans*, *RP.initialState.robots*)
12: **until** *RP.children* ≠ NULL
13: **return** *RP*

Algorithm 6 GETNEXTBEST()

```
1: while ¬fringe.IsEmpty() ∧ goalnode ≠ NULL do
2:   node ←REMOVE-FIRST(fringe)
3:   if GOAL-TEST(node) then
4:     goalnode ← node
5:   end if
6:   fringe ←INSERT-ALL(EXPAND(node)) {assign next robot to all possible tasks}
7: end while
8: if goalnode = NULL then
9:   return NULL {all allocations were tried}
10: end if
11: allocation ←CREATE-ALLOCATION(goalnode)
12: return allocation
```

4.4 Evaluation of Allocations

As mentioned in Section 4.2 the search nodes in the fringe are sorted descending by their possible utility value. The calculation of these values and everything concerning the evaluation of search nodes and allocations respectively is explained in this section.

Every plan has its own criteria to decide if a task allocation is good or not. For example an attackplan needs a robot with a precise shoot and a defendplan needs a fast and big robot to block the enemy. Therefore every plan has its own utility function. These utility functions are able to evaluate search nodes or in other words incomplete task allocations and complete task allocations with respect to the criteria of their plans. A utility function in terms of this project work has the following structure:

$$\mathcal{U}_p = w_0\, pri + w_1\, f_1 + \ldots + w_n\, f_n$$

There are several summands, which consists of a function f_i and weighting coefficient w_i. f_i is an arbitrary function mapping the capability of task allocations in respect of one criteria to [0..1], whereas 0 is bad and 1 is good. The following examples are the first implemented summands and also part of this project work.

Priorities The first summand pri is a special summand, which is part of every utility func-

tion. It captures the preferences roles have towards tasks.

$$priorities = \sum_{r \in Robots} pref(Role_r, Task_r)$$

Distance to the ball An elementary question is, which robot should dribble the ball. The almost simplest answer is, that the robot which is next to the ball should take that task. That means for the function of this summand respectively criteria - the shorter the distance, the better.

$$distBall = \max_{In(r,p,Attack)} 1 - \frac{dist(r, ball)}{fieldDiagonale}$$

$In(r, p, Attack)$ denotes that robot r is assigned to the task Attack in plan p.

Distance to the own goal This summand has almost the same function as the last one. The difference is that the robot, which maximise the function, has to be close to the own goal instead of the ball.

$$distOwnGoal = \max_{In(r,p,Defend)} 1 - \frac{dist(r, ownGoal)}{fieldDiagonale}$$

Similarity to old allocations In a cooperative multi agent soccer teams it is important to agree and avoid oscillating about the task allocation. Therefore the so called similarity summand was implemented. If an allocation has to be changed, this summand benefits allocations which are similar to the last one.

$$similarity = \sum_{r \in Robots} sim(OldTask_r, NewTask_r)$$

The similarity summand influences the result of the utility function in a different way than the other utility summands. The similarity summand can only decrease the result according to the differences between the old and the new task allocation. If both task allocations are complete the same, the similarity summand evaluates to zero.

Another precaution, which takes care of the same issue as the similarity summand, should be mentioned here. If a robot enters a state the first calculation of its task allocation tree is done with respect to the allocated tasks of the other robots in this state. In other words, an additional robot avoids to change the allocations of its teammates. This is achieved by

creating initials search nodes which already have the teammates allocated to the tasks they are currently executing.

To implement utility functions that cause the robots to behave as desired it is important to choose the right weighting coefficients. For example the weighting coefficients of the similarity summand could dominate all other summands and as a consequence it would be impossible to calculate other allocations than the current one. Another negative example occurs if the priority summand is weighted to low. In such case the robot with the goal keeper role and its special abilities could be allocated to the attacker task.

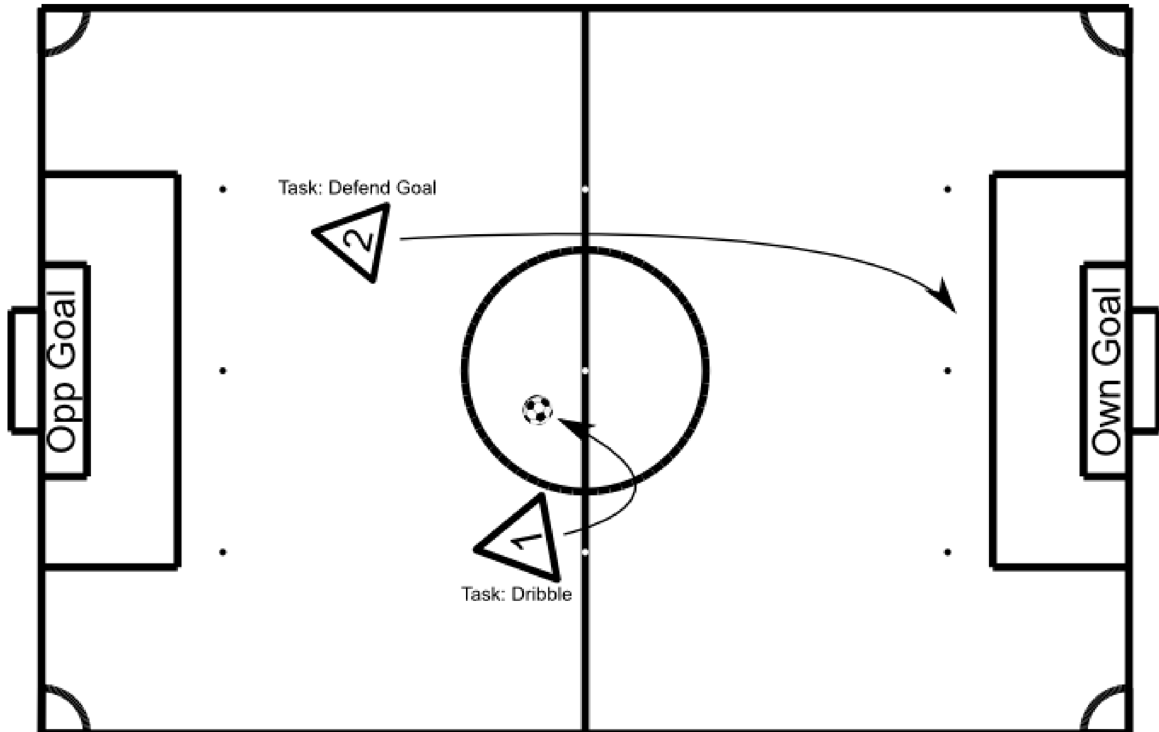


Figure 4.4: Utilityfunction example

Figure 4.4 shows a positive example. In this example the weighting coefficient of the ball distance summand is greater than the weighting coefficient of the own goal distance summand. Robot 1 is also closer to the ball than robot 2, therefore robot 1 does not defend the own goal although it is closer to the own goal than robot 2. To gain an idea how the weighting coefficients influences the robots behaviours it is absolutely necessary to test different configurations beyond the simulator.

5 Evaluation and Future Work

5.1 Evaluation

This project work was evaluated under simulator conditions as well as under laboratory conditions and tournament conditions - both with real robots. In this chapter, a small extract of the evaluation is described. One part is the performance test. This performance test described the time interval needed to calculate the most complex task allocation tree ever used by the Carpe Noctem RoboCup Team. This tree is shown in Figure 5.1. The tree is for 5 robots and has 6 levels, which mostly consists of plantypes with just one plan. The exception is level 4 with two plans in a plantype. Each plan has an utility function with an average of 3 utility summands.

The performance test was run hundred times. The minimum elapsed time was 0.5895ms [1] and the maximum elapsed time was 0.6781ms. The average elapsed time was 0.6259ms with an standard deviation of 0.0192ms.

In Chapter 6 of Triller [10], the disagreement about task allocations was evaluated. This evaluation of disagreements partially evaluates this project work, too. For example, the similarity summand and the initial search nodes with already allocated teammates (Section 4.4) benefits the mutual consent of the team about task allocations. Another point is that the performance of the PlanSelector has direct influence on the time to come to a mutual consent about the task allocations. Figure 5.2 shows the number of different task allocations over time. The average time to agree on one allocation is $177ms$ with an average deviation of $89ms$. At this point it is important two know that the communication frequency is 10 Hz and that the computation cycle frequency is 30 Hz. Take this into account, it takes $65ms$ to come to know about the difference of task allocations in the team.

[1] 1 millisecond $\hat{=}$ 10000 ticks

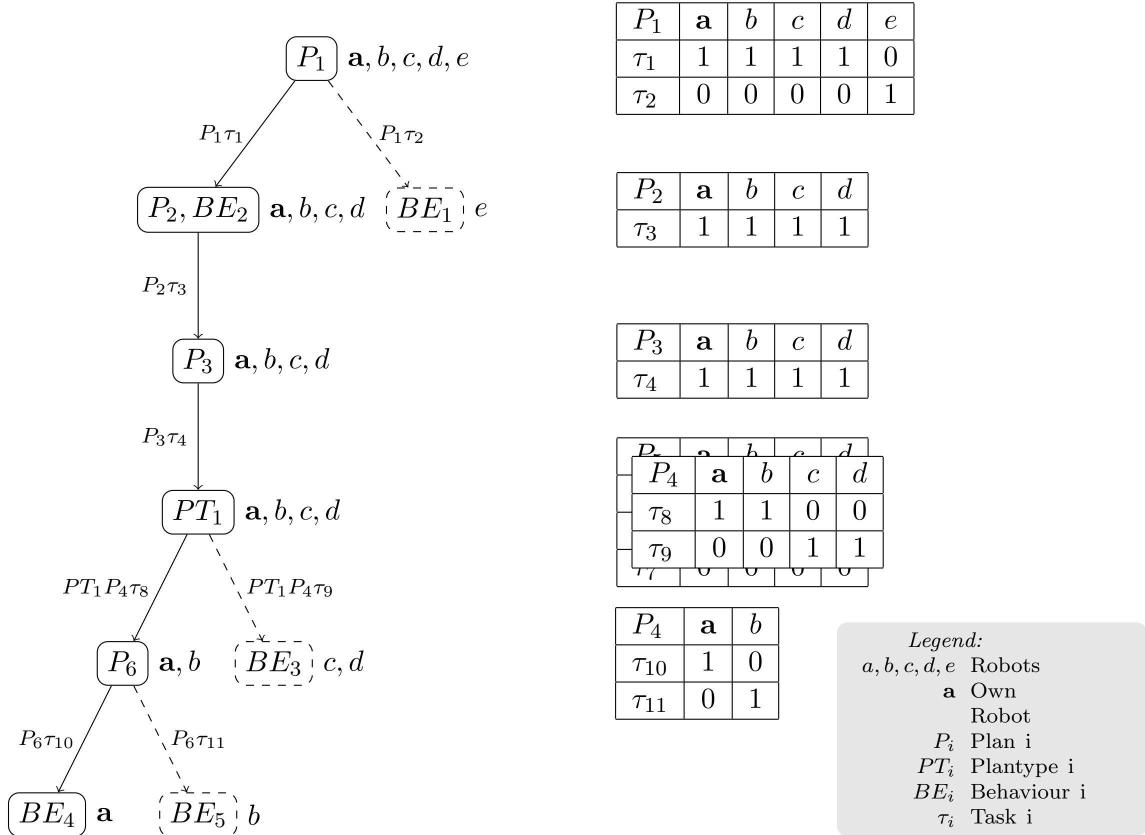


Figure 5.1: Taskallocation Tree for Evaluation

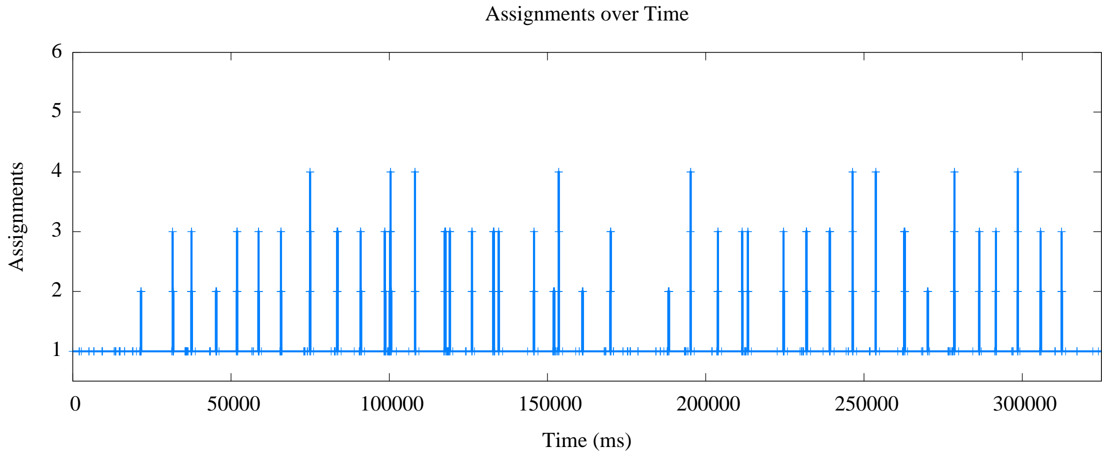


Figure 5.2: Simulator: 6 Robots in a 5min Game with 2 Kickoffs (Start and Half Time)

5.2 Future Work

In this section, a list of ideas is given, which could improve the performance or provide an ease of use of the PlanSelector.

Weightening coefficients As mentioned in Section 4.4 it is difficult, but also important to choose the right weightening coefficients for the utility summands. Machine learning algorithms could take over the tuning of these coefficients.

Comparability of utility functions Each plan has its own utility function with different utility summands. If the worst result of a utility summand is 0.8 it probably dominates all other utility summands. The influence of this summand can be decreased by its weightening coefficient, but the weightening coefficient must be adjusted every time a new utility summand is implemented. A method which constructs preference relations based on fuzzy sets could deal with this problem.

Improve plan selection The PlanSelector always chooses the plan whose criteria are best possible fulfilled. If this plan cannot be successfully executed because of a reason which is not taken into account, this should be noticed. The robots could collect data during a game to avoid those problematic plans.

Learning criteria The robots have to collect data about the situations and about the results of the executed plans during a match. Based on this data it is possible to extract new criteria, which have to be fulfilled for a successful plan execution. The collected data could also be used to learn, which plans are effective against certain teams. Maybe this could even be done during a match.

6 Summary

The RoboCup initiative provides football tournaments for autonomous mobile robots with highly dynamic environments. In this environment it is necessary to base the robots decision making on efficient and robust mechanisms. This project work addresses the problem of efficient decision making about strategic plan selection and task allocation in highly dynamic environments.

The foundation for the implementation presented in this project work is ALICA – A Language for Interacting Cooperative Agents. ALICA specifies how robots use plans to solve problems as a team. This project work is the first implementation of ALICA-based decision making algorithms and is integrated in the behaviour engine of the Carpe Noctem Software Architecture.

This project work decides which plan of a plantype is executed and how the tasks of this plan are allocated to team members. The decision is made by comparing the results of the additive utility function of each plan. The underlying algorithms are based on the A* search algorithm.

Considering the requirements for a efficient decision making the performance of the developed implementation is more than adequate. It was successful used at the RoboCup GermanOpen 2009 in Hannover and will be used at the RoboCup World Championship 2009 in Graz.

Bibliography

[1] Philipp A. Baer. *Platform-Independent Development of Robot Communication Software*. PhD thesis, University of Kassel, Germany, 2008.

[2] Gero Bisanz and Gunnar Gerisch. *Fußball: Kondition- Technik- Taktik... Und Coaching*. Meyer + Meyer Fachverlag, 2008. ISBN 3898993140.

[3] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a. *Journal of the ACM*, 32:505–536, 1985.

[4] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4 (2):100–107, 1968. doi: http://dx.doi.org/10.1109/TSSC.1968.300136. URL `http://dx.doi.org/10.1109/TSSC.1968.300136`.

[5] Shigeo Hirose and Edwardo F. Fukushima. Snakes and strings: New robotic components for rescue operations. *International Journal of Robotics Research*, 23:341–349, 2004.

[6] D. Rus, B. Donald, and J. Jennings. Moving furniture with teams of autonomous robots. *Intelligent Robots and Systems, IEEE/RSJ International Conference on*, 1: 235, 1995. doi: http://doi.ieeecomputersociety.org/10.1109/IROS.1995.525802.

[7] S. J. Russell and Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.

[8] Andreas Scharf. Grafische Verhaltensmodellierung kooperativer autonomer Robotersysteme. Bachelor thesis, University of Kassel, Germany, 2008.

[9] Hendrik Skubch, Michael Wagner, and Roland Reichle. A Language for Interactive Cooperative Agents. Technical report. University of Kassel, 2009.

[10] Stefan Triller. A Cooperative Behaviour Model for Autonomous Robots in Dynamic Domains. Diploma thesis, University of Kassel, Germany, 2009.