VERTEILTE SYSTEME

UNIKASSEL VERSITÄT

A Cooperative Behaviour Model for Autonomous Robots in Dynamic Domains

University of Kassel

Diploma Thesis from

Stefan Triller

At

Distributed Systems

Reviewer: Prof. Dr. Kurt Geihs
Prof. Dr. Albert Zündorf

Supervisor: Dipl.-Inf. Michael Wagner
Dipl.-Inf. Hendrik Skubch

January 6, 2009

Erklärung

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Diplomarbeitarbeit habe ich bisher bei keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Baunatal, den 06.01.2009

Abstract

Team work in teams of autonomous robots is getting more and more important in various domains. This thesis presents a software that is capable of controlling the individual and team behaviour of such robots. It is the first implementation of the language ALICA, which allows specifying team behaviour. Individual autonomous behaviours are encapsulated in team plans. Although those team plans are specified by humans, with ALICA, there are build-in points in the specification where the robots can take autonomous decisions on which concrete plans to execute. Therefore the robots communicate their internal states with each other so they are aware of what their team members are doing. The result is an implementation of ALICA that will be used in the domain of RoboCup for autonomous football robots.

Contents

List of Figures

1 Introduction

Coordination within a team of cooperative autonomous robots is a research topic around the globe. Robots act autonomously on their view of the environment, keeping a "team goal" in mind which they try to achieve. Strategies realising coordination usually need some kind of communication in-between all participating actors. Robustness against unreliable communication and sensory noise is difficult to achieve and must be taken into account when designing a software framework for teams of robots. Examples for dynamic domains where cooperative behaviour is needed are rescue robots who try to make their way through a disaster scenario to save humans, military robots that jointly clear a field of mines, or robots playing football.

The last example was picked up by the RoboCup Federation[1], whose goal is to develop a team of fully autonomous humanoid robots that can win against the human world champion team in football, by the year 2050.

Football is a very dynamic domain, as positions of team members, opponents and the ball are changing frequently, so the robots need to react quickly on the changing situations. Besides the challenges of localisation, detecting the ball, opponents and controlling the individual behaviour, their team goal is to win the football game.

The Distributed Systems Group at the University of Kassel successfully takes part in the RoboCup competitions with a Middle Size Team called *CarpeNoctem*[2]. The team consists of six robots whose behaviours need to be coordinated. Some robots should defend the own goal and one should kick the ball into the opponent's goal.

This thesis targets at developing a software that is capable of controlling the cooperative actions of such RoboCup robots. This control software should not be limited to the domain of

[1] http://www.robocup.org/overview/241.html last accessed 01/02/2009
[2] http://carpenoctem.das-lab.net last accessed 01/02/2009

RoboCup and should allow the execution of behaviour, previously defined by human developers. This behaviour is specified according to the newly defined language ALICA (Skubch et al. [27]). With this language it is possible to specify plans that describe how a team of robots should act together. The control software of each robot parses a machine readable representation of such a description and executes it. While executing these plans, the robots communicate their internal states to each other and hence are aware of what other team members are doing. The knowledge about what plans other team members are executing allows each robot to make its own decisions on which plans it executes. Within these plans the robots autonomously decide for specific tasks, in order to achieve the goals of the plans together. Hence the coordination within a team of robots is the core of this thesis.

This work is organised as follows: Chapter 2 describes the foundations, which represent the background of this thesis. Chapter 3 describes the problem of coordinating a team of autonomous robots in detail. The following chapter (Chapter 4) explains the model of the implementation. In Chapter 5, the implementation for the model, shown in Chapter 4, is explained. An evaluation is given in Chapter 6, and Chapter 7 sums up the thesis and outlines future work.

2 Foundations

This chapter deals with the basic foundations that are necessary to build multi-agent systems. At first a description of agents is given, which is then extended by a description of multi-agent systems. Some existing multi-agent systems are presented in Section 2.3. Section 2.4 introduces a new language for specifying team behaviour for autonomous systems which builds the base for the implementation presented in this thesis.

2.1 Agents

First of all, a definition for the term *agent* in the context of computer software should be given here. Unfortunately there is no unique definition for the term *agent*. There is a general agreement among researchers that the term *autonomy* is important for its definition. But the problem is that software agents are used in various domains equipped with different abilities, so there are different definitions for them as well (Wooldridge [29]). Wooldridge [29] (page 17) states a definition as follows:

"An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives."

Basically an agent uses information about the environment from its sensors and acts based on them. This is usually an ongoing interaction. If the agent modifies the environment with its actions, this leads to different information on its sensors again. Figure 2.1 illustrates this.

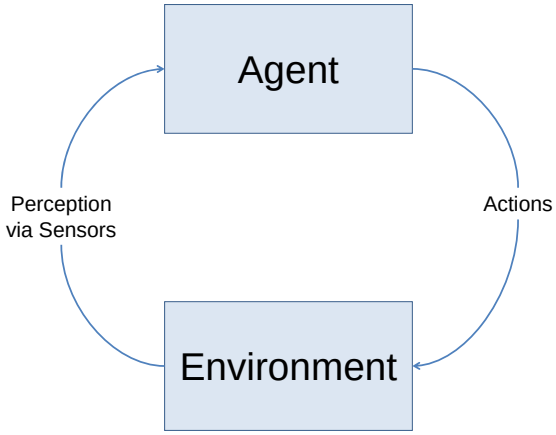


Figure 2.1: Agent and Environment Interaction

Wooldridge and Jennings [30] associate the following properties with agents:

Autonomy Agents that have the ability to operate on their own, without any human intervention.

Social ability Agents that are able to communicate with other agents (or even humans) by using a communication language.

Reactivity Reactive agents perceive data from their environment and respond to changes that occur in it.

Pro-activeness Pro-active agents keep a special goal in mind and try to achieve it by *taking the initiative*.

Shoham [25] points out in his paper about *Agent Oriented Programming (AOP)* that agents, like humans, have knowledge, beliefs, intentions, and obligations. Bates [2] even associates emotions with agents.

An agent need not necessarily possess all of the properties mentioned above. Different agent types may have different abilities. Nwana and Heath [20] attempt to classify agents and give an overview about different types. These types are:

Collaborative agents Autonomous agents that cooperate due to their social ability.

Interface agents Autonomous agents with the ability to *learn*.

Mobile agents Mobile Agents do not move physically in terms of driving or walking, but are able to move their code, including its execution state, on to another processor.

Information/Internet agents Internet search engines or crawlers.

Reactive agents Systems with the *reactivity* ability, mentioned above.

Smart Agents Smart Agents combine the abilities of being autonomous and cooperative. They are also able to learn.

Hybrid agents Systems that cover more than one type.

Based on different types of agents, Wooldridge [29] distinguishes between four agent systems:

- Deductive Reasoning Agents
- Practical Reasoning Agents
- Reactive Agents
- Hybrid Agents

Deductive reasoning agents use a *symbolic* representation of their environment and desired behaviour. These symbolic representations are *logical formulae* and they are manipulated using *logical deduction* or *theorem-proving*. There are two problems to solve in such a system. At first, one needs to find a way to properly describe the real world environment in an adequate symbolic description, and secondly the agent needs to be able to reason about (manipulate) these in an adequate time span.

Bratman [4] defines a practical reasoning system as:

"Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires / values / cares about and what the agent believes."

In contradiction to theoretical reasoning, which is directed towards beliefs, practical reasoning is directed towards actions.

A reactive agent system is a totally different approach compared to approaches that use symbols and logic. Researchers thought it is not possible to build agents using symbolic/logic systems that are able to operate in time-constrained environments. Brooks [5, 6] states that intelligent behaviour can be generated *without* explicit symbolic representations and *without* explicit abstract reasoning. He also says that "intelligent" behaviour arises through

the interaction of the agent with its environment. The basic idea behind a reactive agent system is that it directly maps from situations to actions without reasoning.

Hybrid agent systems are systems that combine reactive and pro-active behaviour. An example for such a system would be an autonomous vehicle driving between two cities on a road populated by other vehicles. A reactive layer of the system with it's "situation - action" - mapping could avoid colliding with obstacles. The proactive layer could be the one navigating the vehicle to the other city.

2.1.1 BDI Agents

Belief-Desire-Intention (BDI) systems are procedural reasoning systems (PRS). They are first mentioned by Bratman [3] who investigated the human way of thinking and planning. He divides the human way of thinking into three groups:

Beliefs Beliefs represent the state of the environment for the agent. They are *uncertain* information about what happens around it. It is also possible to have beliefs about beliefs.

Desires Desires are goals and/or sub goals for the agent. They represent the objectives or situations the agent wants to accomplish or bring about.

Intentions To accomplish its desires, the agent has a "database" of plans. The agent selects plans out of this database. A plan contains the instructions to achieve a goal. Active plans, or plans in execution are referred to as intentions.

Figure 2.2 illustrates a procedural reasoning system. Its interpreter reasons about the beliefs and what plans are available. It then selects a plan that can fulfil a certain desire and adopts it as an intention.

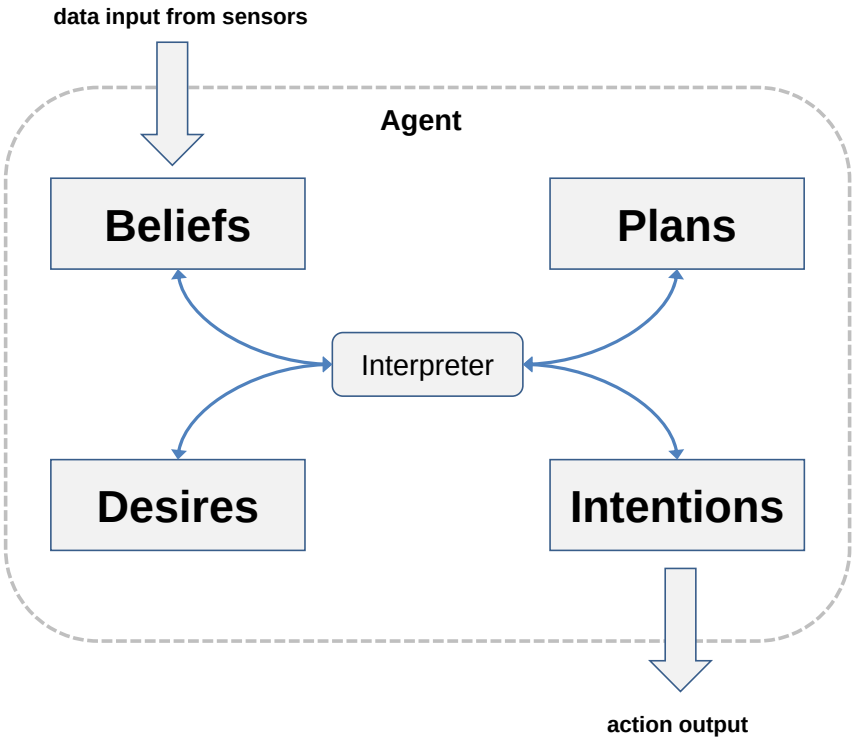


Figure 2.2: The Procedural Reasoning System (PRS)

A PRS was first implemented by Rao and Georgeff [22].

2.2 Multi-Agent Systems

A multi-agent system contains a number of agents that interact with each other using some form of communication. Communication can either be explicit or implicit. With explicit communication, the agents actively send data to each other. Implicit communication can occur because of changes in the environment. If one agent changes the environment, another agent may react on this change. Likewise in single agent systems, those agents are able to interact with their environment, but not necessarily share the same sphere of influence. Spheres of some agents may coincide, i.e., more than one agent may have control over it or is able to influence it.

Figure 2.3, adapted from Jennings [16], illustrates such a multi-agent system.

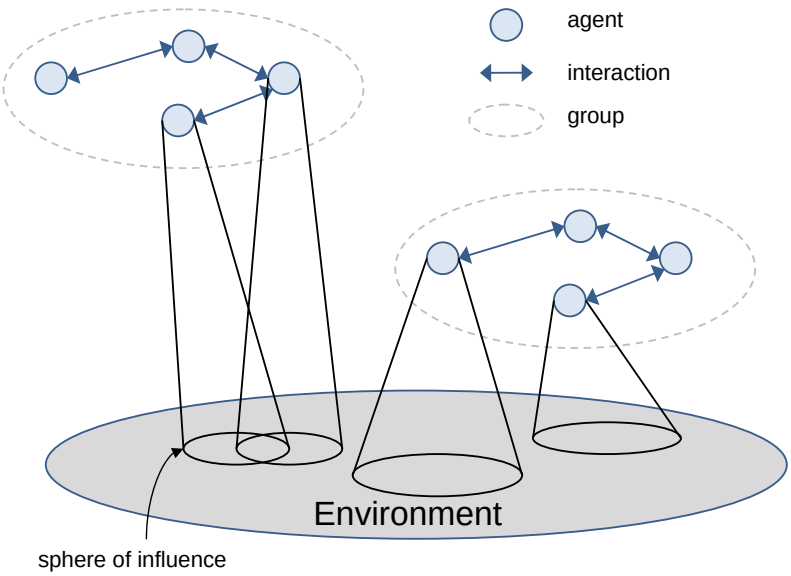


Figure 2.3: Typical Structure of a Multi-Agent System

In a multi-agent system, in order for an agent to achieve its goals it might be dependent on actions of other agents. Sichmann [26] and his colleagues distinguish between four different forms of such dependency relations.

Independence All agents are independent from each other and thus there is no dependency between them.

Unilateral One agent depends on another, but not vice versa.

Mutual Both agents depend on each other while trying to achieve the same goal.

Reciprocal dependence Agent A depends on agent B for some goal, while agent B depends on agent A for some goal. These two goals do not need to be the same.

An implementation for a multi-agent system should consider these dependencies in its design and offer possibilities for agents to make agreements. The following section explains two theories for collaboration in multi-agent systems, *joint intentions*, and *shared plans*. It also contains a short survey of existing implementations of multi-agent systems.

2.3 Related Work

This section describes two theories for collaboration in multi-agent systems and compares some existing implementations of such systems with the software developed in this thesis.

2.3.1 Joint Intentions

Levesque et al. [18] investigated how people do something together in a group. They state that the *joint action* of multiple people involve more than just the union individual actions, even when they are coordinated. A good example, they mention in their paper is ordinary automobile traffic. There is no *team work* involved, although drivers act simultaneously and are coordinated by traffic signs and rules. If a group of drivers decides to drive somewhere together as a convoy they need to communicate with each other about their individual intentions. If they would not communicate their intentions with each other and only some of them know the way to their destination, some might get lost if others drive too fast. Another situation is that if one driver stops, the others assume that they have reached their destination. But that need not be the case and thus leads to the necessity of communicating their individual intentions. This human approach is adapted to agents and a definition of a joint intention by Levesque et al. [18] is:

"A joint commitment to do an action while mutually believing (through the execution of the action, that is) that the agents are doing it."

This definition makes clear that whenever a group of agents wants to achieve a goal together, they have to keep in mind what the others are doing. If one agent for some reason does not believe that the goal can be achieved, it needs to communicate it to the others, so they will stop executing their tasks. Every time one of the agents discovers some circumstances that affect the possibility of reaching their goal, it needs to establish a mutual belief among the team about the new circumstances. In addition, besides notifying others about what action an agent commits to, it is essential to have reassuring confirmations to make sure the mutual belief does not dissipate over time (Cohen and Levesque [7]).

2.3.2 Shared Plans

Shared plans, introduced by Grosz and Sidner [12], is one theory about multiple agents that act together. The idea behind it is that not all agents within a group need to have full

knowledge on how to achieve their team goal. Since BDI agents have a plan repository (see Section 2.2), they can solve certain problems using the plans within it. If a problem arises which cannot be solved by one single agent within a group of agents, it needs to ask for help. By definition of a shared plan, group members have adequate knowledge about each other's capabilities (Grosz and Kraus [11]). Because of this knowledge it is possible for an agent, if it is not able to execute a plan, to delegate it to other members of its group, which have the capabilities to execute it. All plans, which are necessary to solve a certain problem and which are spread upon all group members, can be seen as a *shared plan*. In order to establish such a shared plan, it is essential to establish a mutual belief among all group members, about who is doing what to achieve the common goal. For establishing such a mutual belief, the agents need to communicate with each other. Communication is necessary, because other group members have to be aware of what a single agent's capabilities are and to which plans it is committed. This is needed to determine if the common goal can be achieved with all available capabilities and plans. Agents within a group, trying to achieve a common goal do not necessarily need to have the same intention to achieve it, but they cooperate to achieve the goal. Since agents may have multiple goals, including some they cannot achieve alone, such cooperation is essential.

2.3.3 Existing Implementations of Multi-Agent Systems

Within the topic of multi-agent systems, several models of how to describe their behaviour have been developed. Some of which are similar to the implementation provided in this thesis are outlined in this section.

Schurr et al. [24] describe an implementation called *Machinetta*, which is based on *STEAM* (Tambe [28]). STEAM again bases on *SOAR* [17], an architecture for single agents. STEAM is a domain independent implemented teamwork model. Operators in SOAR that express a single agent's activity are enhanced by *team operators* in STEAM. Team operators explicitly represent a team's joint activities. The team establishes a *joint intention* to execute this operator. Like in SOAR, STEAM uses a hierarchy of these operators to describe a complex behaviour by combining small activities into a tree of activities that is able to manage complex tasks. In STEAM the lowest operators in this hierarchy are individual operators that are executed on a single agent. STEAM operators on higher levels may be team operators.

Each team member maintains a private state for itself to execute individual operators and a

team state to apply team operators. With the help of these maintained states it is possible to establish a *joint intention* with multiple agents dynamically and to react on failures of a team member or other unforeseen events. In addition, STEAM adopted some parts from the *shared plans* theory, the sharing of an agent's capabilities with its team members, together with its current beliefs to achieve a mutual belief. The Machinetta approach on top of STEAM is to have a proxy software that represents the agent. This makes it possible to even have heterogeneous teams of humans and robots working together, because one could have a PDA running that proxy software to coordinate one's behaviour with the robots.

The plans presented in this thesis are also designed for multi-agents with individual sub-tasks for individual agents, but allow for autonomous decisions for an agent to take. Agents are not tied to execute one specific plan under a certain condition, but may select the most suitable one for the current situation, out of a given set of plans.

MOISE$^+$ (Hübner et al. [15]) together with its implementation, written by Hübner et al. [14] is also based on a hierarchical structure, like STEAM. It is a tree with a global goal as its root. Below the root node, there are sub-goals, which need to be achieved to reach the global goal. Sub-goals are plans that can have the following relations to each other: sequence, choice, or parallelism. Plans in sequence need to be executed in order. If there is a choice point within the tree, the agent may choose one of the associated plans. Parallel plans are executed together. The description of a team uses roles instead of agents directly. The concept of roles in MOISE$^+$ supports derivation, for instance in soccer the attacker role is a specialised player role. In addition to roles, there are groups that may represent a sub-team and contain the necessary roles with the possibility to specify the cardinality for each role. Groups also may be compatible with each other so an agent playing a certain role in one group can be allowed to play another role in a compatible group. Within (between agents) and between those groups, three different types of *links* exist: authority, communication, and acquaintance, meaning one agent may control another, the agents can communicate and an agent is allowed to have a representation of another agent, respectively.

Relations of plans to each other like parallelism, choice, and sequence and the concept of using roles rather than agents directly, is adapted in this thesis. Roles are split up into roles and tasks in this thesis, because it allows for a better separation of the two different concepts. Roles should reflect an agent's capabilities and abilities, whereas tasks specify what an agent should do in certain situations.

In the RoboCup domain, the *German Team* (Düffert et al. [9]) in the former *Four Legged*

League (Sony Aibos) used a behaviour specification language called *XABSL* developed by Lötzsch et al. [19]. XABSL basically is an XML format that allows specifying the behaviour of a single agent. It represents a rooted directed acyclic graph of options. The terminal nodes within this graph are *basic behaviours* that execute simple actions like driving to a point and are implemented in C++, independently from the XML specification. Within the options of this graph, the activation of basic behaviours on lower levels is realised via state machines. Each state has a subsequent option or subordinated basic behaviour. Besides this, each state has a decision tree with transitions to other states. Decisions are made based on the agent's world state and environment data. In the Middle Size League of RoboCup, the team from the University of Stuttgart (CoPS) uses XABSL to represent conditions on transitions in their Petri Nets for specifying team behaviour (Zweigle et al. [31]).

The idea of these basic behaviours and a language on top of it to coordinate them with a state machine is very similar to the software presented in this thesis. Petri Net plans as specified in Zweigle et al. [31] do not allow for autonomous decisions of what plan to execute next. It is a predictable state machine with the same outcome on the same situations, which is changed in this thesis.

Our requirements for a language to describe team play are:

- It should provide an easy way to model guided and autonomous decisions for agents, i.e., humans can develop *plans* for agents.
- It should allow for quick reactions and autonomous decisions by each robot confronted with a highly dynamic domain.
- The break-down of a robot should be compensated on the fly.
- The unreliability and potential cost of communication, i.e., estimate decisions made by all other agents, should be taken into account.
- A plan for a team of agents should allow for sub-tasks for individual agents and the agent should select the most suitable one.
- Autonomous selections of a plan out of a group of plans should be supported.

Approaches so far do not meet all requirements. Skubch et al. [27] describe them in more detail and compare them to existing approaches. A description of the new language *ALICA* (A Language for Interacting Cooperative Agents) is given in the next section.

2.4 ALICA

ALICA uses a hierarchical state machine with n-hierarchical levels to represent the behaviour of agents. This concept enhances the reuse of modelled behaviour, because common decisions based on the environment knowledge of the agent that lead to a change in its behaviours can be made on higher levels. Special cases with more concrete environment conditions can be handled on lower levels. Team behaviour is explicitly taken into account in ALICAs design. ALICA offers a global perspective on the team while modelling its behaviour. This thesis is the first implementation of ALICA and the following two sections will explain the syntax, semantic and rules of this language.

2.4.1 Syntax

The syntax of ALICA defines the following elements:

Plan A plan is a set of states, connected by transitions. Every plan has a utility function as well as pre-, runtime-, and post-conditions. It also contains a set of tasks and parameters.

Plantype A plantype is a set of plans.

Behaviour A behaviour is a description for a basic behaviour. A basic behaviour is always a single agent program. It is an atomic element and does not allow for interaction with other agents.

State A state may contain an arbitrary amount of behaviours and plantypes. It is connected to other states by transitions. Because states are a part of plans and states may contain plantypes, a hierarchy of plans emerges.

Transition A transition connects exactly two states, *S1*,*S2*, with each other and is bound to a condition. If this condition holds, an agent in state *S1* progresses along the transition to state *S2*.

Synchronisations Synchronisations connect several transitions with each other to ensure that agents progress along those transitions simultaneously.

Task A task annotates a part of a plan. Each such plan-task pair is associated with a cardinality. One task may be attached to several different plans. It not necessarily needs to describe the same plot line on all plans.

Role A role has characteristics that describe the capabilities of an agent.

Utility A utility is a weighted sum of arbitrary influence factors based on an agent's belief state.

In addition several syntactical constraints exist, such as:

- A state always belongs to just one plan.
- A transition connects exactly two states, which need to be within the same plan.
- Synchronisation only occurs within a plan.
- The hierarchy of plans must not contain cycles.

2.4.2 Semantic

This section describes the semantic meaning of the elements introduced in the previous section.

Plan A plan is a recipe for one or more agents that describes how to achieve a certain goal. Due to the attached tasks on a plan, an agent chooses a part within that plan. This means that the plan is modelled for a team of agents, but not every agent enters every state within the plan. With a precondition, the choice of the plan and task can be restricted, e.g., a precondition could require the possession of the ball. In the same way, runtime-conditions can be formulated – they need to hold during the complete duration of the plan execution. Post-conditions describe the result of a plan. This means, what conditions hold after the plan has been successfully executed. These post-conditions build the foundation for planning in future work.

Plantype Theoretically a plantype can group arbitrary plans, but in practise it is useful to group only plans with similar post-conditions. A plantype could contain several attacking plans: attack over the left flank, attack over the right flank, and attack through the middle of the field. The target of all three plans is to get into a good position to shoot a goal. At runtime, exactly one plan out of this plantype is selected according to utility functions.

Behaviour A behaviour can be seen as the smallest, atomic plan. It cannot contain states, but it may have pre-, runtime- and post-conditions.

State All plantypes and behaviours within a state are executed in parallel. If a state contains a plantype, a plan out of it is selected and executed.

Transition Transitions specify the sequence of states. Their attached conditions tell when an agent changes from one state to another.

Synchronisations With synchronisations, arbitrary transitions within a plan can be connected. Before agents may change from one state to another, where the transition in-between is part of a synchronisation, an explicit synchronisation through communication is required. This ensures that two or more agents reach their target states at nearly the same time.

Task A task is a plot line within a plan. One or more agents, depending on the cardinality of the plan-task pair, commit to a task and afterwards execute one part within the plan.

Role The characteristics of a concrete agent are matched onto requirements of a role which later on abstracts from that specific agent. A role can be taken by various agents that match the required characteristics. Additionally a role contains priorities for tasks. These priorities, together with arbitrary summands of the utility function, describe a preference, of an agent having this role, towards committing to such a task.

Utility A utility function determines the quality of a plan with respect to a certain situation. Its summands can be chosen arbitrarily, based on the agent's belief base. For example, one summand could take into account the distance between two agents.

2.4.3 Rules

One part of ALICA's semantic are rules that describe the changes of an agent's internal state. The internal state of an agent is represented by all plans in execution. Because of the nesting of plantypes and states that may contain plantypes again, this results in a tree structure. Every node in this tree represents a specific plan, together with an active state and an assignment describing what agent is executing which task within this plan. The root of this tree is called *top-level plan*.

This section describes the rules of ALICA.

Init-Rule The Init-Rule denotes the start of an agent's execution with its top level plan.

Trans-Rule An agent changing from one state to another uses this rule. It evaluates the precondition of a transition and if it holds, the agent changes its state along the transition to another state.

Alloc-Rule This rule is closely related to the *Trans-Rule*, because if the newly entered state contains plantypes there needs to be an allocation of tasks to the available agents for a plan selected from this plantype. The *Alloc-Rule* specifies that an agent commits to one task in a plan and executes the associated sub-plans and behaviours. Sub-plans and behaviours are those that are in the active state of a given plan.

STrans-Rule The *STrans-Rule* (Synchronised Transition) establishes a mutual belief before two or more agents are allowed to change their states. They need to synchronise through explicit communication.

BSuccess-Rule Describes a success of a behaviour, marks it as successful and stops it.

PSuccess-Rule Describes a success of a plan, marks it as successful and stops it together with all sub-plans and behaviours.

If a failure occurs while executing a plan or behaviour, ALICA handles them with the following repair rules.

BAbort-Rule A failed behaviour is marked as failed and its execution is stopped.

BRedo-Rule Describes the default reaction if a behaviour fails. The failed behaviour is restarted, if possible. This rule may be overridden by the *Trans-Rule* that leads the agent to another state.

BProp-Rule If a failed behaviour was restarted by the *BRedo-Rule* and still results in a failure, the failure is propagated to the parent plan.

PAbort-Rule Marks a failed plan as failed and stops it together with all its sub-plans and behaviours.

PRedo-Rule Restarts a plan, if possible, with its current assignment if it fails for the first time.

PProp-Rule If a failed plan was restarted by the *PRedo-Rule* and still results in a failure, the failure is propagated to the parent plan.

PTopFail-Rule If the top-level plan of the agent fails, it will be retried by the *Init-Rule*.

PReplace-Rule May override the *PProp-Rule*. Instead of propagating the failure to the parent plan, this rule tries to find a new assignment for the failed plan to satisfy the plan's conditions, or if the plan was selected from a plantype, another plan with a new assignment.

NExpand-Rule Handles a failure in task allocation. If all possible assignments for a plan are tried and if none is able to satisfy all conditions, the plan is marked as failed.

Replan-Rule Periodically checks the utility of a task allocation, and triggers a new task allocation if the current utility is deemed to be unsatisfying.

After explaining the foundations of this thesis, the next chapter describes the problem addressed in this thesis and gives an overview about autonomous robots in general. Special attention is given to the domain RoboCup, because the software developed in this work is evaluated in this domain.

3 Problem Definition

This chapter describes domains where teams of autonomous robots can be used and explains why team play is becoming more and more important. Special attention is given to the RoboCup domain, because this work is evaluated in this domain. Section 3.4 gives an overview about the current software architecture of the CarpeNoctem RoboCup team and how this thesis will be integrated. Finally, Section 3.5 describes the task definition for this thesis.

3.1 Cooperative Teams of Autonomous Robots

Cooperative teams of autonomous robots are used in various domains. One domain is rescue robots that are able to rescue humans after a catastrophe such as an earthquake. In this scenario the robots should clear the areas where it is too dangerous for humans to operate in order to rescue people. Robots could then form a team and split up over the area to search for people or ask for help from another robot, or even a human in a heterogenic team of robots and humans.

Robots exploring other planets like they are now exploring Mars is also a domain for a team of autonomous robots. Having a team of robots with different abilities, one could survey the planet whereas another robot could analyse samples of the soil. If the survey robot has found an interesting place to take samples of the soil, it can tell its team member to take this task.

A team of cooperative autonomous robots could also build a house together. One of them might have the ability to lift heavy parts such as a packet of roof tiles up to the roof. Some other team members might be able to establish walls. A single agent would not be able to build the house, but a cooperating team has the required abilities to build it.

To advance research in the field of cooperative autonomous robots, the RoboCup Federation was founded where robots play football in a team. Because this thesis focuses on the RoboCup as its evaluation domain, it will be explained in Section 3.2 in more detail.

3.2 Domain: RoboCup

RoboCup[1] is an international competition for teams of researchers around the world to improve robotics. It was founded by the RoboCup Federation with the main goal to provide an example domain and competition for researchers. Their secondary goal is to have humanoid robots wining against the human world champion in football by the year 2050. Every year, there is one world championship, which always takes place in a different country and several smaller national competitions, like the *RoboCup German Open*[2]. The reason for choosing football as a domain where researchers can develop their robot hardware and software for is that it is a very dynamic and unpredictable, yet simple game. Situations are changing very often and the robots have to adjust themselves to it. It offers a wide variety of research fields, e.g., image processing, communication, cooperation, and artificial intelligence in general. The rules are simplified FIFA rules[3]. There are several leagues: Small Size League, Middle Size League, Humanoid League, Rescue League, Standard Platform League, and a Simulation League. Each league addresses different challenges and therefore has other rules.

Because the implementation of this thesis will be used in a Middle Size team, some rules for this league are explained here. A team consists of six robots that need to make autonomous decisions and should act together in order to win the game. The field size is 12m x 18m, the robots are no larger than 50cm x 50cm x 80cm and they play with a standard red FIFA winter ball.

Localisation is usually done via a camera and image processing software that calculates the robots position based on the white field lines. Every year the rules are changed to become closer to the original FIFA rules, for example in the past the field was smaller (9m x 12m), the goals had different colours (yellow and blue) and there were only four robots per team. Communication between robots is done via a standard wireless network connection; they share their position and the location of the ball for example. Based on their world model that represents the state of their environment, they decide how to behave in order to reach

[1] http://www.robocup.org last accessed 01/03/2009
[2] http://www.robocup-german-open.de last accessed 01/02/2009
[3] http://www.fifa.com/worldfootball/lawsofthegame.html last accessed 01/02/2009

their common goal.

3.3 Team Play

Team play is becoming a more and more important factor in the RoboCup Middle Size league, as the field size increases. A pass might help in a situation where the robot that possesses the ball is surrounded by opponents, because it could pass it to a team member that waits in a free area. Besides this pass there are other huge advantages of team play, e.g., the robots can split up to cover wide ranges of the field in order to defend their own goal, block the middle field, or start a cooperative attack. A team without the ability to coordinate their individual behaviour might run into problems such as: Covering only one small area of the field with all their robots or having two team members fighting for the ball resulting in loosing it to the opponent. Strategy and tactic is another advantage of team play over individual acting robots. For instance, it is possible to dynamically swap positions of a defender and an attacker, to start an attack out of the own half. Within a team, robots can also replace each other in case one drops out, e.g., if the goal keeper drops out a field player can take its task.

Other cases where team play is important are standard situations. Namely there are: Kick-off (start and restart of the game), Throw-in (after the ball has crossed the field border), Goal kick (when the opponent team has shot the ball behind the field line next to the goal), Free Kick (happens after a foul was committed by the opponent team), and Corner kick (when the own team kicks the ball behind the field line next to their goal). The RoboCup rules require that those situations cannot result in a direct goal, so the ball has to touch at least one other robot besides the one scoring the goal.

In order to make team play possible the robots need to communicate with each other or use strategies such as plan recognition. Until now the robots of the team *CarpeNoctem* communicate some of their world model data and make their decisions based on them, e.g., they assign roles assuming the other robots hopefully will calculate the same result out of the same data.

3.4 Integration into the Existing Framework

The implementation presented in this thesis will be used within the CarpeNoctem Middle Size RoboCup team. This section explains how it is integrated into their software framework. The *CarpeNoctem Software Architecture* consists of several modules, each running as a single process, making it more robust against software failures. Interprocess communication is realised using the network protocol UDP. In this way, every module can be programmed in any programming language matching the needs of its specific task.

The *Base* module, written in C#[4] can be seen as the "robots central unit". It contains an *Ego World Model* holding the robots own environment data, a *Shared World Model* holding data from its team members and the *BehaviourEngine*, controlling the actions of the robot. Figure 3.1 gives an overview about the architecture. The robots sensors are the *Compass* Module and the *Vision* Module. The *Vision* Module is written in C++ and is responsible for localisation and detecting objects like opponents, ball, and the goals. This data is integrated into the *Ego World Model*. The *Compass* is necessary because the robot needs to know in which direction of the field the opponent's goal is located. The robot's actions in the environment are realised with the *Motion* and *Kicker* Modules. They communicate with the robot's hardware and receive or send information from or to the *Base* Module.

All modules can be started remotely using the program *LebtClient*, which is able to send Start-/Stop-Commands via wireless network to the *Lebt* Module that starts or stops the modules accordingly. The *Lebt* Module also provides information about the robots internal state to the user who monitors the *LebtClient*. Communication between the robots and the different modules is based on Spica (Baer [1]), a framework that abstracts from communication protocols, sockets and even the programming language used on each robot. Section 5.4.1 explains how the robots use Spica for communication in detail.

Every box in Figure 3.1 represents a module and the arrows in between mark their interaction. The *Base* Module contains the two world models and the BehaviourEngine.

The following subsection describes the current BehaviourEngine inside the *Base* Module in detail, points out its disadvantages and defines the task of this thesis.

[4] *CarpeNoctem uses the Open Source implementation of C#: Mono (http://www.mono-project.com last accessed 12/24/2008)*

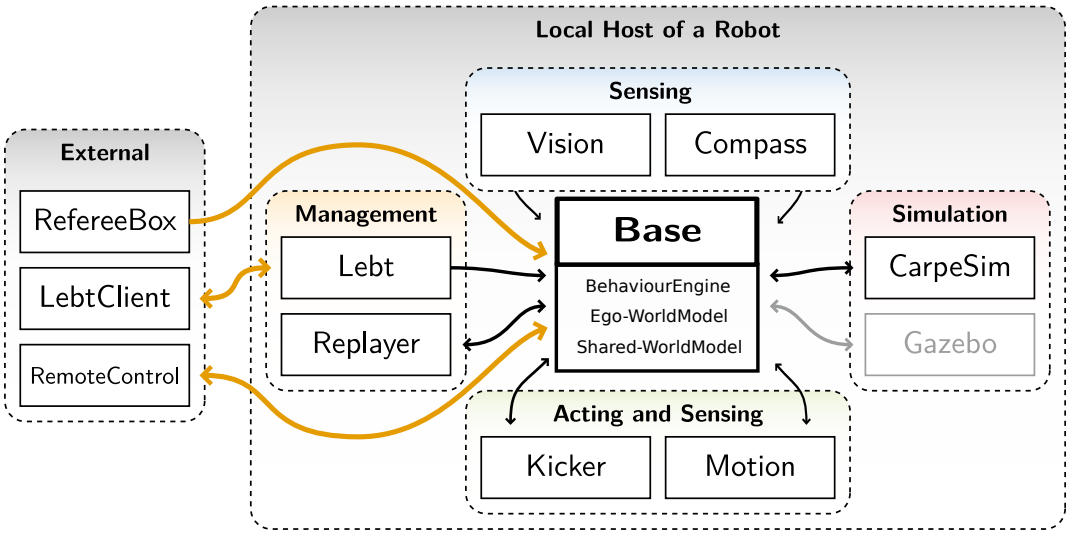


Figure 3.1: CarpeNoctem Software Architecture

3.5 Task Definition

The goal of this thesis is to replace one part of the *Base* Module in the *CarpeNoctem Software Architecture*, the *BehaviourEngine*. The BehaviourEngine controls the execution of *Basic Behaviours*. *Basic Behaviours* are small programs started as a thread within the *Base* module, sending out messages to the *Motion* Module or *Kicker* Module. These messages contain commands, which will result in a movement of the robot or a kick by its kicking device. *Basic Behaviour* threads are usually executed with a frequency of 30Hz with a few exceptions explained in Chapter 5.

The current BehaviourEngine is a two level hierarchical state machine. Its top level consists of *Globals*, wherein transitions are defined that are executed, no matter in what context the robot is in inside the lower level. The lower level consists of *Policies* which are changed by transitions defined in the *Globals* definition. Policies contain *Contexts* and contexts contain *Basic Behaviours*. Figure 3.2 gives an overview about the architecture.

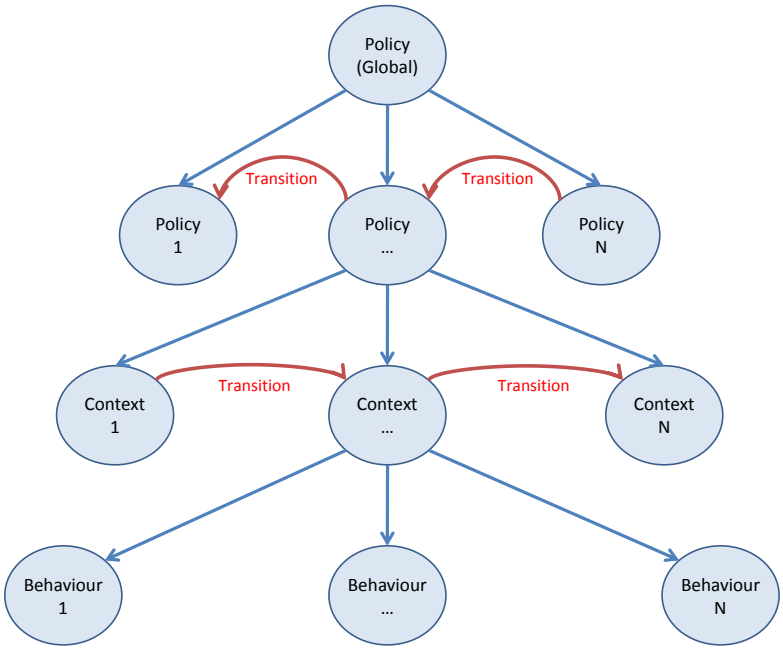


Figure 3.2: Levels of the Current BehaviourEngine

Listing 3.1 shows the content of a *Globals* definition. The initial policy in line 2 defines that the state machine will start with the policy *Wait*. Transitions are defined inside the "[Transitions]" block with a name of a transition on the left side of the equation and the target policy on the right. For example the transition "Start" will lead to the policy "Game6Pol".

```
1  [Game6]
2          Initial Policy = Wait
3
4          [Transitions]
5                  Start = Game6Pol
6                  RStart = Game6Pol
7                  RKickoff = Kickoff
8                  RKickoffOpponent = KickoffOpponent
9                  ...
10                 Stop = Wait
11         [!Transitions]
12 [!Game6]
```

Listing 3.1: Contents of a Globals Definition

Listing 3.2 illustrates the content of a policy definition. A policy may contain several *Contexts* and each context can have one or more *Basic Behaviours* running in them. The *Basic Behaviours* may have optional parameters defined in a "key = value" format. It is not possible for a robot to be inside more than one context at a time. Line 2 defines the start context for the robot entering this policy, "Attack". Inside the Attack context it will execute the following basic behaviours: AttackDribble, GoalKick, RoleAssignment, and Observer. Below these basic behaviours in Line 28 the transitions between contexts are specified. Line 28 describes that if the transition "Defender" is triggered, the robot being in context Attack will switch to context Defend.

```
1  [Game6Pol]
2          Initial Context = Attack
3          [Attack]
4                  [AttackDribble]
5                          Deferring = 0
6                          Frequency = 30
7                  [!AttackDribble]
8
9                  [GoalKick]
10                         Shot Radius = 6000.0
11                         ...
12                         Deferring = 0
13                         Frequency = 30
14                 [!GoalKick]
15
16                 [RoleAssignment]
17                         Deferring = 0
18                         Frequency = 10
19                 [!RoleAssignment]
20
21                 [Observer]
22                         WatchBallLost = true
23                         ...
24                         Deferring = 0
25                         Frequency = 30
26                 [!Observer]
27
28                 Defender = Defend
29                 Supporter = Support
30                 DefendSupporter = Defend
31                 AttackSupporter = LineMan
32                 Keeper = KeeperNoBall
33                 WithoutTask = WithoutTask
34
35                 BallLost = Wander
```

```
36          [!Attack]
37          [Defend]
38                  ...
39          [!Defend]
40          ...
41  [!Game6Pol]
```

Listing 3.2: Contents of a Policy Definition

The current BehaviourEngine has the following disadvantages:

1. Behaviours trigger transitions.
2. RoleAssignment is implemented as a behaviour.
3. Team play was not considered while designing it.

Before explaining the biggest disadvantage of the current BehaviourEngine, the lack of team play, points 1 and 2 are described because they are necessary to understand the problem.

The problem with behaviours triggering transitions is that two concepts are mixed: The logical part, like, reasoning about world model data and the basic action part, e.g., sending commands to the robots kicker or motion. To react on a certain condition, it is necessary to put a basic behaviour into the context where the condition should be monitored. This makes basic behaviours depended on the context within they are executed. Examples for such basic behaviours are *RoleAssignment*, *Observer*, and *Timer*. Explicit modelling of conditions on a higher level, instead of using basic behaviours to monitor conditions, would improve the reuse of basic behaviours.

Currently, the RoleAssignment behaviour determines the own role of a robot and triggers a transition (Defender, Supporter, etc.) whenever it changes. Transitions in Listing 3.2, starting from line 28 tell the robot to change its context according to the current role it has. The *Observer* behaviour monitors certain data in the robot's world model and triggers transitions whenever the monitored data is changed. Line 35 of the policy definition defines a transition that is triggered by the Observer behaviour. Finally, the *Timer* behaviour triggers a transition when a given timer runs out.

The biggest disadvantage of the current BehaviourEngine is that team play is not considered explicitly. There is just a rudimentary implementation for standard situations. A robot *waits* for a certain time span and assumes that its team members have done their job when the time span has passed.

For example, on an indirect Free kick, the RoleAssignment behaviour determines an Attacker and Defender, which both are in their according contexts *AttackFreeKick* and *DefendFreeKick* afterwards. To prevent the robots from changing their roles these and the following contexts will not include the RoleAssignment behaviour. The context *AttackFreeKick* contains a basic behaviour to slowly kick the ball to the defender. The defender executes the *Timer* behaviour that triggers a transition when the timer runs out, resulting in a context change into one where it executes a behaviour that catches the ball.

These shortcomings are addressed in ALICA (Section 2.4). Conditions can be specified outside of basic behaviours, i.e. no conditions encoded as transitions inside basic behaviours. The concept of *roles* is improved, as well as extended by the concept of *tasks* to handle for example Free kick situations better than explained above. In addition, ALICA uses a hierarchical state machine with n-hierarchical levels instead of just two.

The task of this thesis is to address these issues and develop a new BehaviourEngine, implementing the rules of ALICA, which offer the possibility of team play. Thus, the new BehaviourEngine should not only be able to reflect the current game behaviour of the CarpeNoctem team, but should be designed to provide possibilities so it can be easily extended.

After describing the task of this thesis, the next chapter deals with the adaptation of ALICA for the implementation of a BehaviourEngine for the CarpeNoctem RoboCup Team. It describes how the elements and semantic of ALICA are realised in software.

4 Adaption of the Language Specification for Implementation

In Section 2.4, the ALICA specification was described. This chapter targets on introducing a model that implements the language specification with the help of a *domain specific language* (DSL)(Deursen et al. [8]). A DSL is, as it names implies, a language that is specialised for one domain. It allows a particular type of problem or solution to be expressed more clearly than pre-existing languages would allow. For example it would be possible to use the Unified Modelling Language – UML (Hitz and Kappel [13]), to express all elements of ALICA. But since UML is very generic it is very hard to implement model checking. For example, constraints that a state belongs to only one plan are difficult to express. With a DSL it is possible to specify all the constraints of a domain that need to hold, and thus to check them.

The realisation of the single elements within ALICA is explained and shown in figures. In the end of this section a short overview about the complete model is given. To keep the model simple, it is necessary to extend the specification with some elements. Every element in the ALICA specification can be expressed with the DSL.

A complete class diagram that shows the realisation of the DSL is attached in Appendix A.

4.1 Plan and State

According to the specification a plan may contain multiple states, but a certain state may only occur in one plan. Figure 4.1 illustrates the relation of a plan and state.

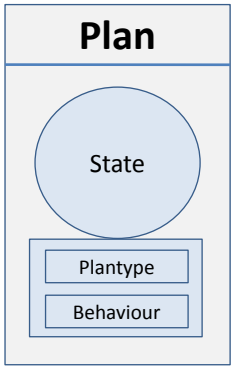


Figure 4.1: Plan and State (with content) in the Model

In addition to states, the implementation of the plan element contains two more elements: *Entry-* and *Exitpoints*. They simplify the model as explained in Section 4.3.

As specified in ALICA, a state may contain the elements *Plantype* and *Behaviour*. In the DSL it is possible to directly execute plans in states. In ALICA this would be represented by a plantype containing just one plan. Through this nesting of plans and states, a hierarchy of plans emerges – the top plan of the hierarchy is referred to as the *Masterplan*. A state in the DSL, likewise as a plan, may contain entry- and exitpoints.

Parameters for plans as specified in ALICA are not implemented yet. This part is planned for the future.

4.2 Plantype

The plantype corresponds to the plantype element specified in ALICA. It is a group of *arbitrary* plans, but as already mentioned before it is reasonable to group those with similar results. Figure 4.2 shows a plantype with several plans.

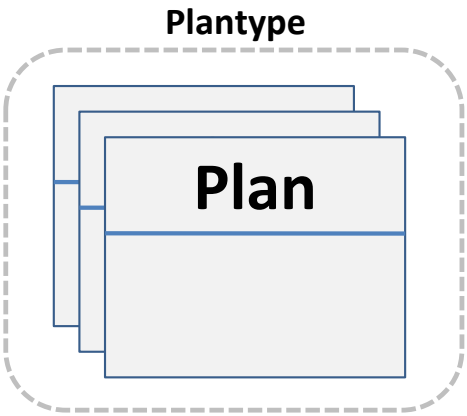


Figure 4.2: Plantype Element

4.3 Connection Points

In the DSL there are two kinds of connection points: entry- and exitpoints. They can be attached to plans or states. Connection points can be connected through transitions. The reason for having connection points is to have a single element in the DSL that allows for attaching transitions to it.

Entrypoints on a plan lead to a subtask within this plan. They have cardinalities in the scope of $[0..*]$, which describe for how many agents these entrypoints are designed.

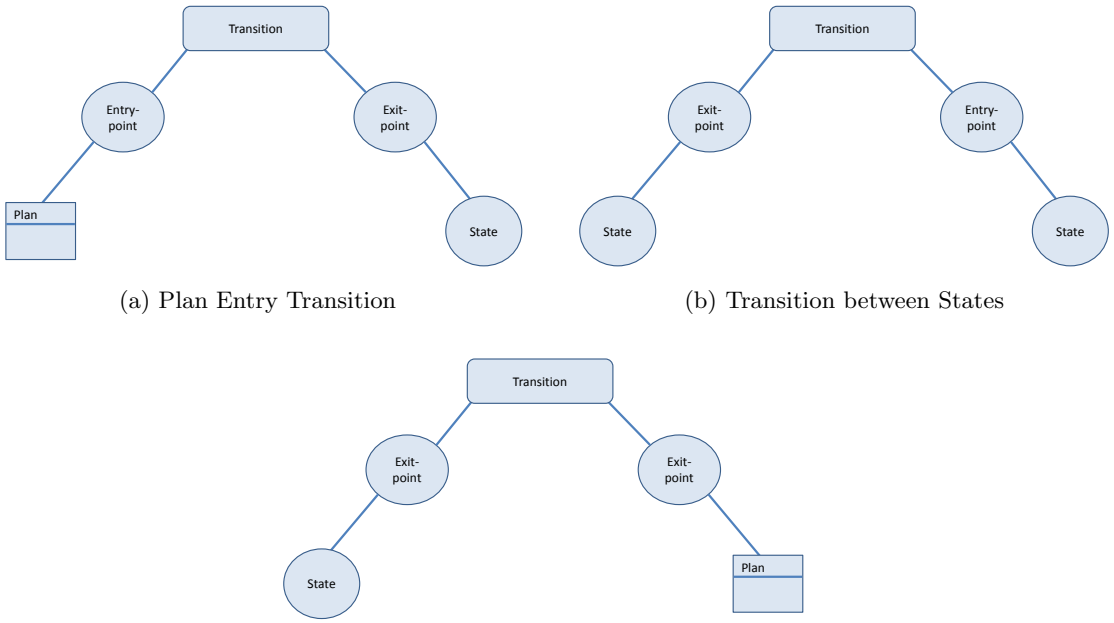
Exitpoints of a plan describe the end of a subtask and can be annotated with a post-condition, independently from each other. They can be classified into success and failure exitpoints and have a result. This way, it is possible to specify which condition holds, for each exitpoint, if the subtask was successfully executed or not.

Connection points that are attached to states do not have a special meaning. They only exist to provide a single element in the DSL where transitions can be attached to.

Related to ALICA, states that are connected by a transition with the plan entrypoint are the equivalent to initial states. Exitpoints of a plan are the equivalent to terminal states.

4.4 Transitions

ALICA specifies a transition as a directed connection from one state to another. The implementation changes the transition element as follows: A transition connects two connection points (entry- or exitpoints). Figure 4.3 illustrates the different combinations.



(a) Plan Entry Transition

(b) Transition between States

(c) Plan Exit Transition

Figure 4.3: Different Transition Combinations

Figure 4.3a illustrates how a transition connects a plan entrypoint with an entrypoint of a state. Figure 4.3b shows how two states are connected to each other by a transition to their connection points. The last case, shown in Figure 4.3c, is a transition that connects the exitpoint of a state with the exitpoint of a plan.

4.5 Synctransitions

Synctransitions realise the ALICA element *Synchronisation*. They connect multiple transitions to enforce an explicit communication between agents, so they reach their target states simultaneously.

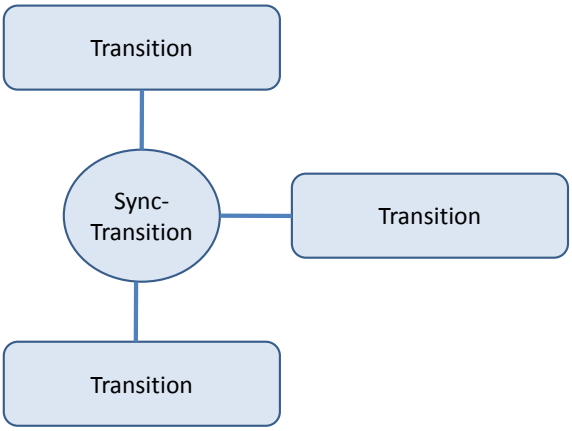


Figure 4.4: A Synctransition Between Three Transitions

4.6 Roles and Tasks

A role has a set of characteristics that describe the capabilities of an agent. Figure 4.5 illustrates that.

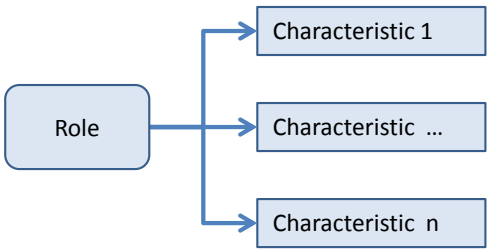


Figure 4.5: Role with Characteristics

In contradiction to the ALICA specification, a plan in the DSL does not have tasks. Tasks are connected to entrypoints (see Section 4.3) in the DSL, which is semantically equivalent to the plan-task pairs in ALICA. The different representation in the DSL is necessary to separate tasks from plans so tasks can be reused on different plans. Figure 4.6 shows this correlation.

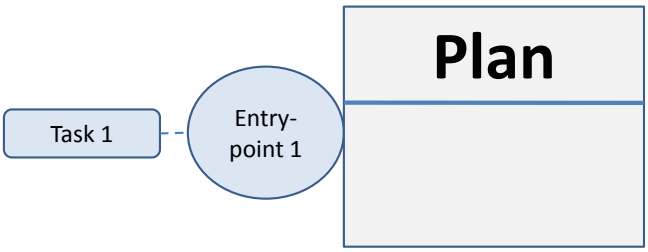


Figure 4.6: Entrypoint with Task

A specific task can be attached to arbitrary entrypoints, even on different plans.

The relation between roles and tasks is realised as follows: The element *RoleTaskMapping* defines a set of "priority - task" pairs for a role. Such a pair exists for every available task and determines the priority a role will commit to it. This makes it possible to pre-define what role preferably commits to which task.

The new element *RoleSet* defines a collection of *RoleTaskMapping*s. It contains the corresponding "priority - task" pairs for every role and tasks of a given Masterplan and sub-plans.

4.7 Behaviours

Behaviours can be parameterised. This is why the DSL has another element besides the *Behaviour* element that describes these parameters – *BehaviourConfiguration*. In this way, behaviours are strictly separated from their parameterisation. These *BehaviourConfiguration*s can be reused in arbitrary states.

4.8 Conditions

Conditions are separated in *Precondition*, *Runtime-condition*, and *Result*. Table 4.1 shows what conditions can be attached to which elements.

Condition	Possible elements
Precondition	Plan, BehaviourConfiguration, Transition
Runtime-condition	Plan, BehaviourConfiguration
Result	SuccessPoint, FailurePoint

Table 4.1: Overview about Conditions

Preconditions must hold at the start of the execution of a plan or behaviour. They also represent the conditions that must hold before a transition triggers. Runtime-conditions must hold during the execution of the plan or behaviour. A result defines the condition that holds if the plan is left on an exitpoint. Figure 4.7 gives an overview about all conditions and where they may be used.

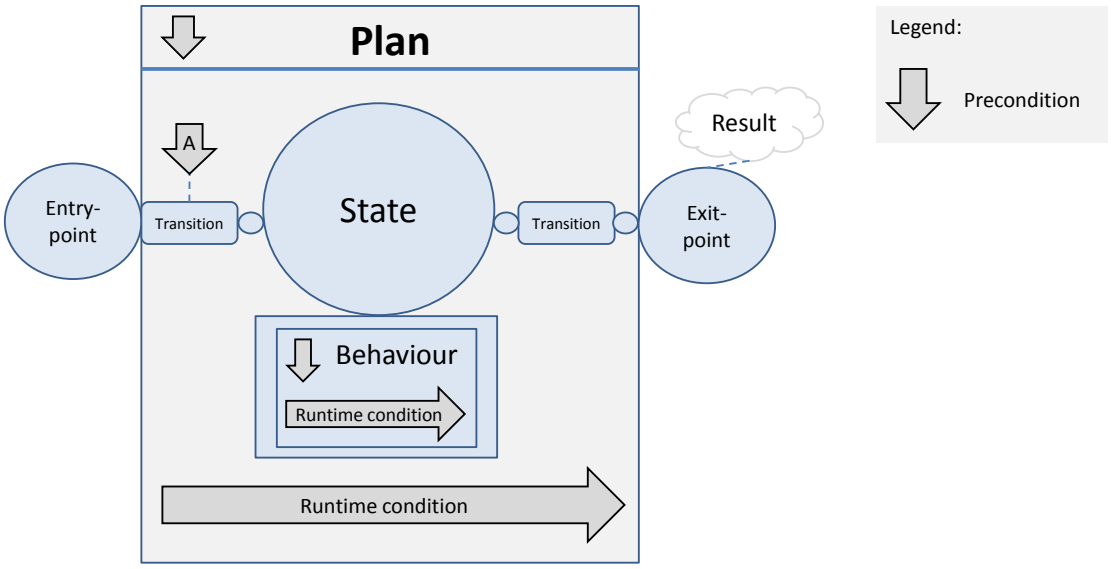


Figure 4.7: Usage of Conditions

Instead of Success- and Failure exitpoints, Figure 4.7 uses only one exitpoint that represents both cases.

These conditions may also use parameter sets on plans. Parameter sets are only supported locally on a plan. They only occur within the pre-, runtime-, or post-conditions of that plan. A mechanism for using parameter sets of parent plans is part of future work.

4.9 Utilities

Utilities estimate how useful their corresponding plan is. At the moment the DSL contains a utility element that consists of a string data type, which describes the utility function. In general, utilities are way more complex and it should be possible to reuse them, so there is room for future work too.

4.10 Additional Elements

During the development of the DSL, some elements are introduced that do not have a direct relation to ALICA. They are introduced to simplify the DSL. These elements are:

PlanElement The highest element in the derivation hierarchy of the DSL. It is used so all elements have an *ID*, *name* and *comment*.

IConnectable An interface between connection points and elements that should be connected through transitions.

AbstractPlan An element that makes it possible to handle all elements within a state in the same way. These elements are *Plan*, *Plantype*, and *BehaviourConfiguration*. They all derive from AbstractPlan.

4.11 Example: Execution of a Plan

This section provides an overview that shows how the elements described in the previous section are used together. Figure 4.8 shows a possible combination of these elements, which describes the cooperation of two agents, *AgentA* and *AgentB*.

Before the *PlanA* shown in Figure 4.8 can be executed, each agent has to calculate an assignment that satisfies the pre- and runtime-conditions and commit to one of the two tasks, *Task1* or *Task2*. This is done considering the following:

- Amount of available agents.
- Cardinalities on entrypoints.
- Conditions A and B on the transitions that are connected to the entrypoints.

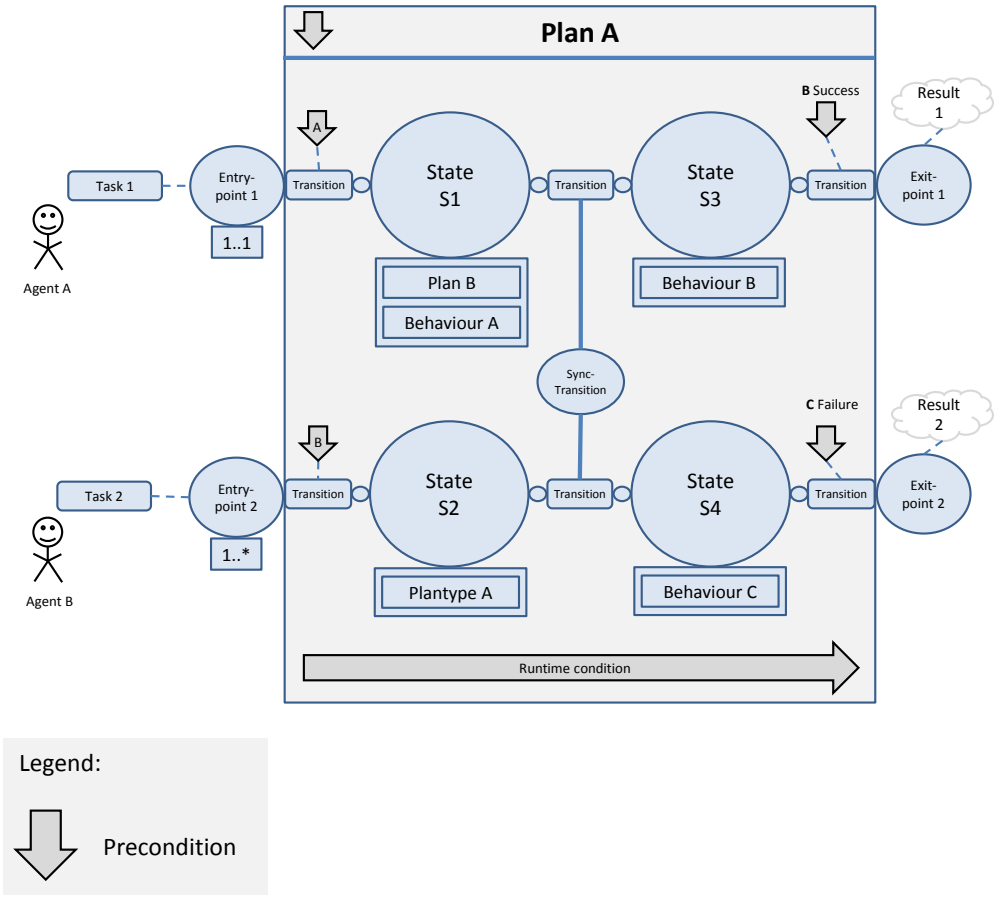


Figure 4.8: Plan Example: PlanA

- Result of the plan's utility function, which considers, e.g., the priority of the role of both agents for the two tasks.

In this example, AgentA commits to the entrypoint with Task1, AgentB to Task2. At first, AgentA enters state *S1* where *PlanB* and *BehaviourA* should be executed in parallel. PlanB is executed as explained for PlanA. BehaviourA is executed directly by AgentA. In parallel to this, AgentB enters state *S2*, selects a plan out of *PlantypeA*, and executes it.

The outgoing transitions of S1 and S2 are connected to a *Synctransition*. This means that the states S1 and S2 may only be left, if both agents have agreed on it, through explicit communication (they have a joint intention, see Section 2.3.1). At this point, it is ensured that AgentA and AgentB are in state *S3* and *S4*, respectively, at the same time. AgentA

executes *BehaviourB* in state S3 and AgentB executes *BehaviourC*.

If the runtime-condition on *PlanA* does not hold anymore, the plan will be aborted.

AgentA leaves *PlanA* with *Result1* if *BehaviourB* is successful. If *BehaviourC* fails, AgentB leaves *PlanA* with *Result2*.

4.12 XMI: An Exchange Format

The persistence format for all instances of DSL elements is *XML Metadata Interchange* (XMI). It is a special form of XML that allows object structures to be made persistent. XMI is a standard of the *Object Management Group* (OMG), so it is supported by various software tools. In this way, it supports the interoperability of heterogeneous software.

For this thesis, this format is used to exchange the plans modelled with the PlanDesigner (Scharf [23]) and the BehaviourEngine that executes them. The BehaviourEngine transforms the XMI files back into object structures that are created by the PlanDesigner. Section 4.13 briefly describes what the PlanDesigner is.

4.13 Behaviour Modeling: PlanDesigner

The language described in Section 2.4 and its adapted version, in Section 4 can be used by a user, to specify behaviour for multiple agents, using the developed tool, named *PlanDesigner*. It is a graphical modelling tool, developed by Scharf [23]. All elements described in Section 4 can be described with it, to model plans for agents. In addition, it supports specification of roles and their priorities for tasks used on plans. They are stored in the exchange format XMI, described in section 4.12. This representation can be parsed and executed by the BehaviourEngine, the result of this work. The BehaviourEngine is explained in the next chapter (Chapter 5).

5 Implementation

In the previous chapter, the implementation model for the *ALICA* specification together with its semantic and syntax was described. This chapter focuses on an implementation based on this model. The implementation for execution of plans created with this model is written in the programming language C#, using its Open source variant *Mono*[1], because most of the CarpeNoctem software architecture is implemented using Mono. The architecture is built in a modular way, so each module can easily be substituted, as long as it implements the according interfaces. A detailed description of the architecture is given in Section 5.2.

Because the previous chapters described the language ALICA for a multi-agent system in general the term *agent* was used to identify a single actor. In the following the term *robot* will be used for an agent, because this work will be evaluated in the RoboCup domain on autonomous football playing robots.

Section 5.4 explains how the robots, executing the BehaviourEngine of this work, communicate with each other. Their communication of the view of the world and internal states to team members is based on the *Spica* framework (Baer [1]).

5.1 Internal State Representation

Before describing the architecture of this engine, the representation of the agent's internal state is explained in this section. Section 2.4.3 explains that the internal state is represented by a tree of plans in execution – *the plan tree*. All plans within the tree are executed in parallel. The root of this tree is called *Masterplan*. Every plan in execution is referred to as *RunningPlan* and has some runtime information attached, such as:

[1] http://www.mono-project.com last accessed 12/24/2008

Plan The plan from the model, explained in Section 4, as specified by the user.

Active State The current state the robot is in within that plan.

Assignment An assignment describes what robot is in which entrypoint of the plan. Being in an entrypoint means that the robot has committed to the task attached to that entrypoint.

Success/Failure Status A status indicator, that indicates whether the plan was successfully executed or failed.

Plan Start Time The time when the robot started executing this plan.

State Start Time The time when the robot entered the active state.

Realised plantype If the plan was selected from a plantype, the plantype is remembered to properly react in the case of a plan failure (see rule *PReplace* in Section 5.5).

An example on how such a plan tree might look like is given in Figure 5.1

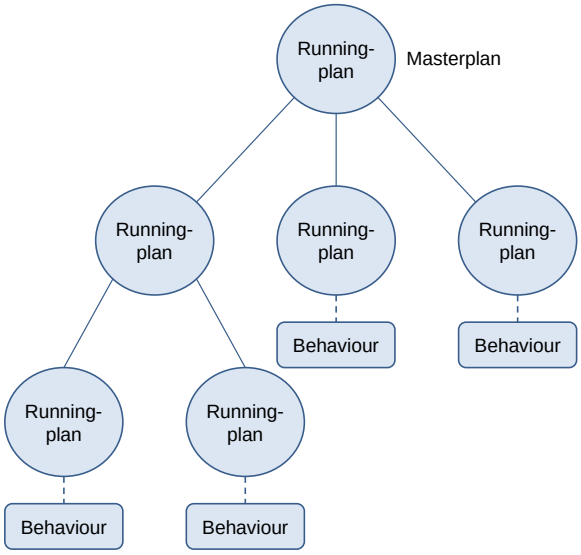


Figure 5.1: PlanTree of RunningPlans

The assignment is the most important part of this internal state representation in terms of team play.

A simplified version of this the original plan tree, referred to as "*SimplePlanTree*" is sent to other team members. A SimplePlanTree contains the *ActiveState* in each tree node together with an *entrypoint*. The combination of a state and entrypoint describes a unique representation of the robot's current position within that plan.

Once the robot has calculated an assignment for a plan (see Section 5.3.5), it will protect this initial assignment for a certain time span[2] from overwriting it with incoming states from other robots. This is done because robots that are supposed to enter the new state, too, might enter it slightly later. If they enter the new state later, they may still send the information that they are in the previous state, leading to confusion. Specifying a time span of when the assignment will not be overwritten does not mean that the clocks of the robots need to be synchronised. The time span is based on the robot's own clock and starts when it begins executing a plan. After this time span, the assignment is updated when a new plan tree from a robot is received indicating that this robot is in another entrypoint than specified in the current assignment.

An assignment is valid, if the following criteria are met:

- The robots in the entrypoints satisfy the preconditions of the entrypoints.
- The cardinality of each entrypoint is met.
- The pre- and runtime-conditions of the plan hold.

5.2 Architecture

The *Base* module of the CarpeNoctem Architecture runs as a single process on the robot, instantiating one monitor thread and one thread for each *Basic Behaviour* of the robot. Each robot is running its own instance of this engine and makes its own decisions based on what it believes about its environment and what it is told by team members. The robots do not send out commands to their team members; they just state a prediction about what others do and assume that they will probably come to the same conclusions based on their commonly shared information. They correct this prediction when they receive information from their team members.

An overview about the architecture is given in Figure 5.2.

[2] Currently, this time span is set to $250ms$, which is 2.5 times of the communication frequency.

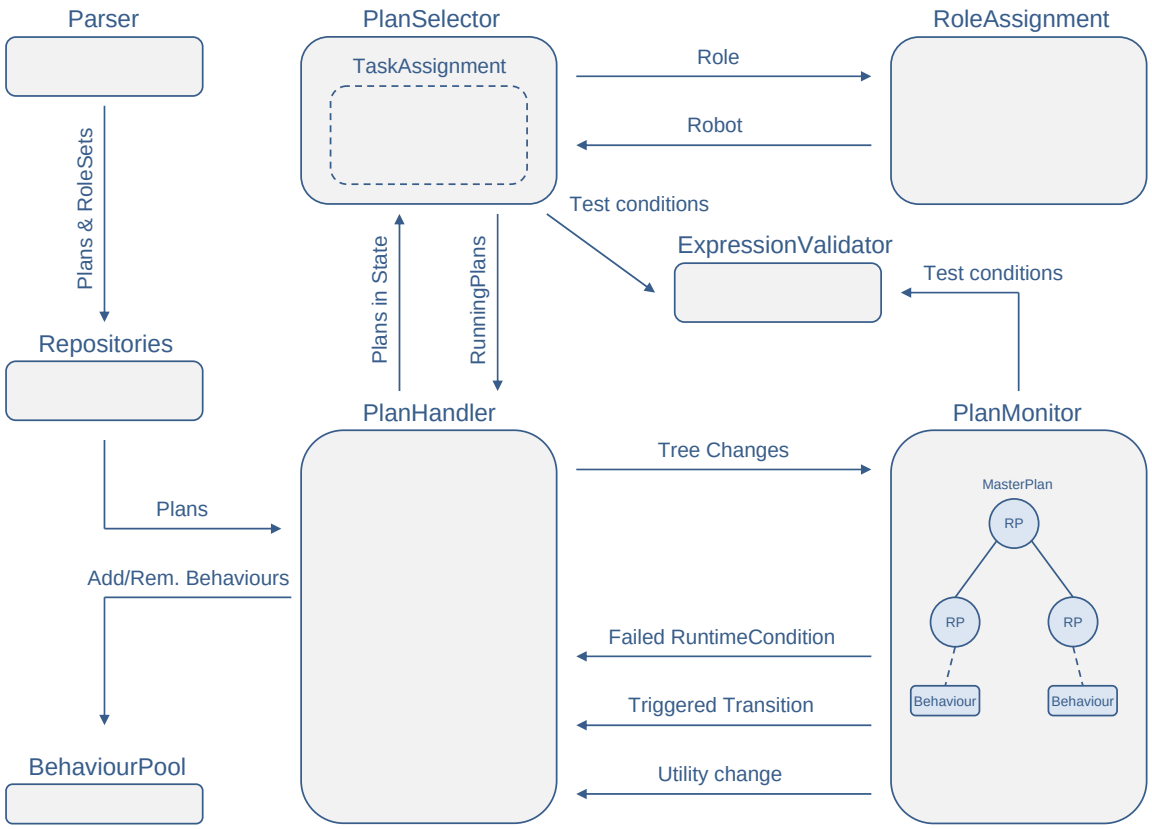


Figure 5.2: BehaviourEngine Architecture

The core of the engine is represented by the *PlanMonitor*, it monitors the current plan tree and notifies the *PlanHandler* whenever a condition for one of the plans changes. Changes in conditions of those plans may result in new plans that need to be executed and therefore the *PlanHandler* builds a set of changes that reflect how the plan tree in the *PlanMonitor* needs to be modified. These two modules are the only ones that directly deal with the robot's internal state.

Whenever an agent enters a new state, the *PlanHandler* calls the *PlanSelector* to build *RunningPlans* for all plans, plantypes, and behaviours inside this state. Basic behaviours returned by the *PlanSelector* to the *PlanHandler* are added to the *BehaviourPool* that executes them as threads. The *ExpressionValidator* is used by the *PlanMonitor* and *PlanSelector* to evaluate conditions.

All modules except for the *PlanSelector* are part of this thesis. An interface to this module

is implemented in this thesis, but its implementation was written by Opfer [21] as part of his project work.

Section 5.3 explains the tasks of all modules in detail.

5.3 Components

Section 5.2 explained how all components of the engine work together. This section describes each module in detail.

5.3.1 PlanMonitor

This module is the core of the BehaviourEngine, it contains the main loop and is started as the monitor thread. The robot's plan tree is also stored within this module.

Inside the main loop, the following events will be monitored. The *Assignment* on each plan is checked whether it is still valid or if one of the assigned robots is in another entry-point, because then it needs to be updated with the new information. Therefore, messages from team members that contain this information are processed. It is necessary to update the assignments to evaluate the runtime-conditions and conditions on transitions with an assignment that reflects the current situation.

The current plan utility value with the current assignment will be calculated and compared with a utility value using all other possible assignments for that plan. If a better one exists, the robot will adopt it and change its task according to the new assignment.

The last function of the *PlanMonitor* module is to evaluate the runtime-conditions of the plan as well as the preconditions of the transitions leaving the current active state in each plan. After each loop the current plan tree is stored inside the robot's *WorldModel*, so it can be send out to team members.

The main loop of this thread runs with a frequency of $30Hz$, but the evaluation of plan utilities is only done every $100ms$ due to performance reasons.

An example for the need of a change in the assignment on a plan could be the following:

Suppose two robots, namely RobotA and RobotB, playing a plan and they are in possession of the ball. RobotA performs the defend task and stays near the own goal. RobotB has the

ball and dribbles towards the opponent's goal. Shortly before the goal, it looses the ball to the opponent. The opponent is faster than RobotB and dribbles towards the goal defended by RobotA. Now it would be a good idea to swap the tasks of RobotA and RobotB, because its team member RobotB is too slow to catch the opponent. The defend behaviour, in the CarpeNoctem implementation, does not actively try to obtain the ball, but positions the robot between the ball and the own goal. In order to steal the ball from the opponent, a robot needs to execute an attack behaviour, which actively tries to obtain the ball.

In this example, RobotA notices that a change in tasks will improve the utility of the plan and performs the change. RobotB receives the plan tree from RobotA, containing the information that RobotA has changed its task and updates his assignment for the plan accordingly. While evaluating the utility for this plan, RobotB should come to the conclusion to change its task too. Now RobotA performs the attack task to attack the opponent driving towards it and RobotB performs the defend task.

5.3.2 PlanHandler

Whenever the plan tree in the *PlanMonitor* module needs to be changed, because of a failed runtime-condition of a plan or a transition, whose condition evaluates to true, a method inside this module is called. It is responsible for changes on the robot's own plan tree and rebuilds the plan tree according to the event that happened.

If a runtime-condition of a plan fails, it is checked whether this plan was selected from a plantype or not. If it was selected from a plantype, a new plan out of the plantype will be tried to handle the plan failure. In case the plan was not selected from a plantype, it is restarted once and if this does not result in success, its parent in the plan tree will be restarted.

A transition can lead to a new state or to an exitpoint of a plan. If it targets a failure exitpoint it indicates a failure of the plan, which is modelled by the user, resulting in the same failure handling described above for failed runtime-conditions. A transition targeting a success exitpoint of a plan results in stopping all plans and behaviours below that plan in the plan tree and marking the plan as success (the success status can be used on an outgoing transition on a higher level in the plan tree).

The last scenario is a transition pointing to a new state, which will result in stopping and removing all plans and behaviours inside the previous state and starting all plans and

behaviours inside the new state.

In order to assign tasks to robots for the plans inside the new state it is necessary to know how many robots will enter the new state. This is done with the other task of this module: Receiving and storing of plan trees from other robots. This module stores the received plan trees, because it is responsible for changes on the robot's own plan tree. A plan tree from a team member may lead to such a change within the plan tree, because of an update of an assignment as described in Section 5.3.1. Plan trees are sent to team members on a pre-defined frequency of $100Hz$. Section 5.4.3 explains this in more detail.

5.3.3 BehaviourPool

The *BehaviourPool* module takes care of starting and stopping threads of *Basic Behaviours*. It loads a library with all available Basic Behaviour classes during initialisation, creates the according objects, and caches them for future use. It also offers access methods for event-based behaviours or behaviours that receive remote commands. Two different types of basic behaviours are supported in this engine; behaviours that are executed if an event occurs and behaviours that are executed at a specified frequency. Event-based behaviours are not triggered with the usual frequency, but executed whenever an event such as a joystick command (for debugging purposes) is received or a ball position is integrated into the robots world model.

This module is called whenever the robot enters or leaves a state that contains behaviours.

5.3.4 RoleAssignment

The module *RoleAssignment* is called whenever the number of team members changes and assigns the roles to the available robots. Roles should be assigned to robots according to their abilities and capabilities, but this is left for future work. This implementation contains a fixed robot to role mapping via a configuration file, but with an event to dynamically trigger a re-assignment. A method performing such a re-assignment can register on this event.

Besides the problem of mapping abilities and capabilities of a robot to the requirements of a role, this module needs to determine how many robots are available. In the current implementation, if a robot does not receive any data from one of its team members for a

certain amount of time[3], this team member will be treated as not playing and thus cannot be assigned a role. There might be smarter ways to determine the amount of currently playing robots, such as using the camera to actually see the playing team members, but this is left for future work, too.

5.3.5 PlanSelector

This module is the core module of the autonomous decisions a robot can make. All plans, plantypes, and behaviours within a state are passed in a list to this module. They need to be executed in parallel. For each plantype in the list, the *PlanSelector* is responsible to select a plan out of it and calculates the best assignment for the selected plan. For each plan in the list, it calculates the best assignment. A plan together with its assignment is called *RunningPlan*.

Since the assignment forces a robot to enter a specific state, this process is repeated recursively until a state only contains behaviours. Behaviours are encapsulated into RunningPlans directly because they are designed for just one robot. A tree of RunningPlans emerges and is returned to the *PlanHandler*.

For selecting a plan from a plantype and its assignment, the utility functions of the plans are evaluated. The robot is aware of how many team members will execute the plantype. The *PlanSelector* searches over the space spanned by plans and possible assignments that satisfy the pre- and runtime-conditions of the plan. The search is guided by a heuristic to estimate the maximum possible utility value and stops if no better than the current assignment can be built. The plan and assignment with the highest utility value will be returned to the *PlanHandler*.

For all plans in execution, this module also checks if there is a better assignment for the plan (see ALICA rule *RePlan* in Section 2.4.3). This check is initiated by the *PlanMonitor* periodically every $100ms$ before it tests the conditions of transitions or runtime-conditions of the plan. To determine if there is a better assignment for the plan, the *PlanSelector* calculates the current utility value for that plan, using the information about the current situation. It searches for a new assignment that has a higher utility value than the current one. During this search the similarity of the assignment is checked to prevent the robots from swapping their tasks frequently. If the difference between the utility value of the current

[3]Currently set to 2 seconds.

assignment and the utility of the new assignment is higher than a pre-defined threshold, the new assignment is adopted.

The BehaviourEngine of this work contains this module, but Opfer [21] developed it, as a part of his project work.

5.3.6 Parser

Plans created by the *PlanDesigner* (see Section 4.13) need to be loaded into the engine in order to be executed. This is done by parsing XML-based files during initialisation of the engine.

The parser works in two stages: Stage 1 creates objects for all elements not specified as a reference, recursively for all plan files used by the *Masterplan*. Stage 2 resolves all references using the objects created in stage 1 and connects them accordingly.

Creating of objects is realised using the factory paradigm (Gamma et al. [10]). The factory paradigm uses an abstract factory class that offers a method to create objects. Concrete factories that build a special object derive from this abstract class and override the create method. This way, one has an interface for creating objects, but lets the sub-classes decide which special object will be created.

Inside the parser, there is a factory for each XML-node representing an element in the model mentioned in Chapter 4. It has two methods, one to create the object (used in Stage 1) and one to attach all children to it (used in Stage 2). The parser can be easily extended by adding new factories. Besides plans it also parses the "role - task priority" pairs (Rolesets) (see Chapter 4) specified in the PlanDesigner.

5.3.7 Repositories

There are two repositories in the engine, one holding all available plans, and one containing the available rolesets. The repositories are filled during initialisation of the engine by the parser module.

At the moment the repositories are only implemented as a storage device with no special access methods. Improving the access methods to to allow for, e.g., filtering plans by a certain condition is left for future work.

5.3.8 ExpressionValidator

All conditions, such as *Preconditions*, *Runtime-conditions* and *Utilities* are automatically generated C# classes and they are loaded during the initialisation of the engine by the *ExpressionValidator* module. The generation of code instead of parsing the expressions at runtime is done due to performance reasons. Expressions might be large strings and it would take a long time to parse them at runtime. Calling compiled auto-generated code is much faster.

This auto-generated code is produced by the *PlanDesigner* (Section 4.13). A class for each plan and behaviour is created that contains the methods to evaluate, in case of a plan, the conditions on the transitions between its states, runtime-conditions, and utility function. In case of a behaviour, a method is created to evaluate its runtime-conditions.

This module loads a library of the compiled auto-generated code and attaches these classes to their matching model elements (a plan or behaviour). Model elements contain an ID and so do their generated classes. For example an auto-generated class for the plan with ID 123 is called *P123*. Attaching the auto-generated code to a plan or behaviour is done during initialisation with *delegates*[4] in C#. Basically, the delegates in the behaviour and plan elements are function pointers and this module sets their targets to the functions inside the compiled auto-generated code. To all other modules in this engine the access to the auto-generated methods is fully transparent and looks like a standard function call.

In this section, all modules of the engine that runs on each robot were described in detail. The following section targets on how a team of robots can play together cooperatively with this engine by communicating.

5.4 Communication Between Robots

Communication is one of the possibilities, besides plan recognition, or tracking of team members, to achieve cooperation between software agents. Although it is not 100% reliable in most scenarios it is powerful because a lot of information can be exchanged at once (depending on the bandwidth of the used medium of course). In this implementation the robots communicate at a frequency of $10Hz$ through wireless network (802.11a or 802.11b). CarpeNoctem uses a multicast approach where all robots can subscribe to a group, send,

[4] http://msdn.microsoft.com/en-us/library/ms173171.aspx last accessed 12/24/2008

and receive data to or from it. The protocol to transport the data is UDP, because the TCP handshake would be too much overhead compared to the size of exchanged data. Resending of lost packets is not necessary, because the environment is changing quickly so old data is mostly useless anyway.

5.4.1 Spica

Spica is a communication framework that offers the possibility to specify what data structures should be communicated, at what frequency this communication should occur, and through what protocol this should happen, in an abstract language. Out of this specification Spica automatically generates sender and receiver stubs for different programming languages, as well as messages, out of the data structures specified by the user. Two of these generated sender and receiver stubs build the communication module for the *Base* module. This communication module offers call-back methods to send and receive data to or from other robots. Commands, such as start and stop from the *LebtClient* control software, are also received via this communication module. All communication passes this communication module, which takes care of serialisation and building objects out of the serialised data, so it is fully transparent to the rest of the engine.

5.4.2 Exchange of World Model Data

Every robot integrates its sensor data into its own world model, where other modules can fetch their information from. Certain data out of this world model is sent to team members, building the shared data of all robots. The BehaviourEngine on each robot takes the shared data into account to coordinate the robots. The current implementation exchanges goal positions, free areas inside the opponent's goal, ball position, ball velocity, and positions of opponents. The ball position contains an additional certainty value, which is used to calculate a shared ball position. For example, the shared ball position represents the position of the ball, most robots agree on. Whenever a robot receives data from one of its team members, it integrates them into its shared world model. The shared world model holds data for every team member and offers methods such as calculating the shared ball position.

5.4.3 Exchange of Internal States

By sending out the internal states of the robots it is possible that whenever two or more robots need to coordinate their behaviour such as changing their tasks in a plan, they know when the other robot has changed its task. Every robot sends out a simplified version of its plan tree (*SimplePlanTree*) to its team members. These SimplePlanTrees contain information about the state and entrypoint the robot is in, for each plan the robot is executing. Assignments on the plans in execution can be corrected using these SimplePlanTrees.

For instance, if the robots take the closest robot to the ball as committing to the task attack, they might come to different results because of the uncertain world model data describing where the robots are. Different results lead to different assignments for the robots, which can be corrected by the exchanged SimplePlanTrees.

5.5 ALICA Rules

This section explains how the rules of ALICA, presented in Section 2.4.3 are implemented in the BehaviourEngine.

The *Init* rule in ALICA is implemented in the same way as described in the ALICA specification, except for ALICA's plan p_0. p_0 in ALICA denotes the root of an ALICA program and consists of only one state, it is the top-level plan. The BehaviourEngine also initialises its plan tree with the Masterplan, but it may contain more than one state.

Changing from state to state in ALICA is realised using the *Trans* rule. Conditions of transitions are monitored periodically during the execution of the BehaviourEngine, along the current plan tree. Whenever a precondition of a transition becomes true, the agent builds a change to its tree. This change represents leaving the state where the transition starts from, i.e., removing its plans and basic behaviours from the plan tree, and entering the new state by adding those inside the new state to the plan tree. This means that the *Trans* rule is implemented as specified in ALICA.

The *Trans* rule is closely related to the *Alloc* rule in ALICA, because if the new state contains plan types there must be an allocation of tasks to the available agents for a plan selected from this plantype. The BehaviourEngine does a task allocation when changing into a state that contains plans or plantypes, according to this specification. An agent

commits to a task and enters the first state after the entrypoint where that task is attached to.

Synchronisations where two or more agents should establish a mutual belief before they are allowed to change their states are covered in ALICA by the *STrans* rule. For the BehaviourEngine this is part of future work and not implemented so far.

The two following rules in ALICA, *BSuccess* and *PSuccess*, describe a success and thus stopping of a behaviour or plan, respectively. The *PSuccess* success rule is implemented as described in ALICA. If a plan reaches a success exitpoint via a transition, this plan is marked as successful and a transition attached to the state where that plan is in may react on it. The plan and all sub-plans as well as basic behaviours are removed from the plan tree and thus stopped.

BSuccess is implemented slightly different to the description in ALICA. A basic behaviour that signals its success (via a member variable in its thread) will not be stopped by default. One can react on it with a transition attached to the state where the basic behaviour is in, as specified in ALICA. The reason for this different implementation is because of old behaviours that are not properly adjusted to the new engine yet.

The rules mentioned before occur during normal execution if nothing fails and everything runs as expected. Of course, there might be failures too. That is why ALICA offers "repair rules", which react on failures that occur.

A reaction on a behaviour failure is described in ALICA's *BAbort* rule. In ALICA, the execution of the failed behaviour will be stopped and it will be marked as failed. In the implementation of the BehaviourEngine, the basic behaviour is marked as failed too, but is not stopped. A transition attached to the state where the basic behaviour is in again may react on it.

The next ALICA rule, *BRedo* restarts the behaviour once, if it fails and no transition reacts on this failure. This is different to the BehaviourEngine, because of the behaviours from the old engine. The behaviour in the engine will not be restarted as in ALICA, but kept running.

Propagation of a failed behaviour is also implemented different from ALICA's *BProp*. A failed behaviour in ALICA is first retried once by the *BRedo* rule and if it still fails, propagated to the parent plan. In contrast to that, the BehaviourEngine marks a basic behaviour as failed. A transition attached to the state where it is in may react on the failure as

specified in ALICA. This is because of the old behaviours again.

Same, as for behaviours, there also exist rules for failures of plans. The *PAbort* rule in ALICA marks a plan as failed, stops it together with all its sub-plans and behaviours, if it fails. The BehaviourEngine also marks the plan as failed and acts according to the following two rules of ALICA, *PRedo* and *Pprop*, if no transition reacts on the failure. *PRedo* in ALICA restarts the plans with its current assignment if it fails for the first time, if possible. The BehaviourEngine does the same; it uses the current entrypoint for the agent and restarts the plan from there.

The *PProp* rule of ALICA will be triggered if *PRedo* was not possible and there is no explicit failure handling modelled by the user. *PProp* then propagates the failure to the parent plan. This failure handling is implemented into the BehaviourEngine, too. It is a recursive construct, so the propagation may go up all the way to the Masterplan. On each level the plan will be retried as specified in ALICA by *PRedo* and afterwards if the failure still persists, propagated to the parent.

A special situation is a failure of the top-level plan. ALICA handles this with the rule *PTopFail*. According to this rule, the top level plan will be retried by *Init*, if it fails, because the agent has no other choice. If a failure of the Masterplan in the BehaviourEngine occurs, the plan tree is reinitialised with the Masterplan, so it acts according to the ALICA rule.

Before the rule *PProp* becomes active in ALICA, it may be overridden by *PReplace*. *PReplace* tries to find a new assignment for the failed plan to satisfy its conditions, or if the plan was selected from a plantype, another plan with a new assignment. On a failure of a plan, the BehaviourEngine restarts the plan once and afterwards it checks if the plan was selected from a plantype. If the plan was selected from a plantype, it tries to replace it with another one from the plantype. The first restart of the failed plan is the only difference to the *PReplace* rule of ALICA in the BehaviourEngine.

If the task allocation for a plan fails, the ALICA rule *NExpand* defines how it will be handled. The plan is marked as failed. This is the same reaction as implemented in the BehaviourEngine, which then reacts on the plan failure again.

The *Replan* rule of ALICA is used to periodically check the utility of a task allocation, and triggers a new task allocation if the current utility is deemed to be unsatisfying. Inside the main loop of the BehaviourEngine, a function checks the utilities of all plans in the plan tree and is able to react if a better assignment can be computed for a plan. This is done by changing the plan tree reflecting the agent leaves the current entrypoint on that plan and

enters the new one. It acts as specified in ALICA.

This chapter described how the language specification of ALICA and its adoption (Chapter 4) is implemented in this thesis. Chapter 6 will evaluate this implementation by comparing it to the old BehaviourEngine of the CarpeNoctem team and pointing out its advantages. Furthermore, it shows runtime measurements and how the engine performs on packet loss and delay.

6 Evaluation

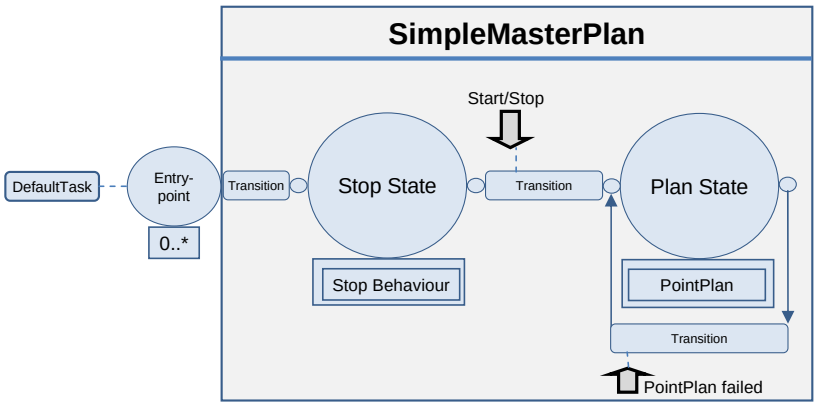
The evaluation is arranged into six sections. At first, an example will be given that shows that plans are able to handle conditions and behaviours are only used for acting. This enhances the re-use of behaviours. Section 6.2 describes how the game behaviour of the former CarpeNoctem BehaviourEngine was modelled with the new PlanDesigner and executed on real Middle Size robots with the engine presented in this thesis. Section 6.4 describes a measurement that expresses how well a team, using this engine, plays together. Finally, Sections 6.5 and 6.6 deal with evaluating the performance of the team behaviour if the network connection is not reliable due to packet loss or delay in simulator tests and on four real RoboCup Middle Size robots.

6.1 Simple Example for Logic in Plans

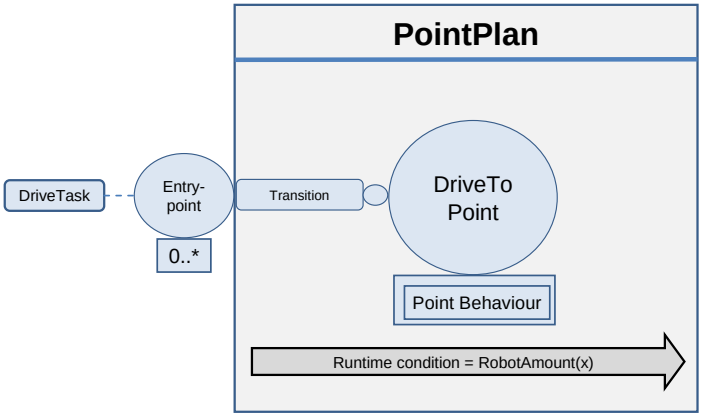
The main goal of this engine is to improve team play and convey the reuse of behaviours. One major feature of this engine which improves team play is the communication of internal states to team members.

The example in this section is simple and shows that a robot knows when a team member joins its plan and reacts on it. The plan is divided into two sub-plans, the Masterplan shown in Figure 6.1a and the plan that contains the logic to react on the joining of a team member, shown in Figure 6.1b.

In this example, robot *A* starts with the Masterplan and automatically executes the stop behaviour in the *Stop State*, because there is no precondition defined for this entrypoint. When it receives the start signal it will enter the *Plan State*. If it receives a stop signal while it is inside the *Plan State*, it will enter the *Stop State* again. The *PointPlan* contains a runtime-condition, monitoring the amount of robots inside that plan. It fails if the amount



(a) Masterplan



(b) PointPlan in Plan State of the Masterplan

Figure 6.1: Simple example for Logic in Plans

of robots in that plan changes. The state *DriveToPoint* contains a behaviour that drives to a specified point, but keeps a distance of $dist = a * 0.8m$, from that point, where a equals the amount of robots inside the plan. Hence, robot A will at first keep a distance of $0.8m$ from that point.

Now robot B start executing this MasterPlan too and receives the start signal. Once it is started, it receives the plan tree from robot A and it also sends out its plan tree to agent A. When it joins the *PointPlan* it knows that robot A is already executing this plan and keeps a distance of $1.6m$ from the specified point.

Robot A received the plan tree from robot B, containing information about that robot B just joined the *PointPlan* and thus its monitored runtime-condition fails. This results in a

failure of that plan. Hence, it will re-enter the state containing the *PointPlan*, according to the model. It also calculates the new distance as $2 * 0.8m$ and thus it drives further away from that point. The same scenario happens if a robot leaves the plan: All remaining robots will re-enter the *Plan State* because of the failed runtime-condition and calculate the new distance.

The amount of robots in a plan is a plan parameter, instantiated by the runtime-condition on its first evaluation. At the moment it is not possible to pass that parameter from the *PointPlan* to the *Point behaviour*, this will be part of future work. For now there is a workaround inside the behaviour. It uses the total amount of robots playing, during its initialisation. This works because it is a test case scenario where the engine of a robot is started or stopped if it should join or leave the plan. Because a behaviour is re-initialised on every re-entering (e.g. if the runtime-condition fails) of a state it is aware of how many robots are left.

The plan parameters allow for building formations, for instance a wall of defence. The defend behaviour has information about how many defenders are available and positions the robot accordingly.

6.2 Comparison with the Former CarpeNoctem Behaviour Engine

Section 6.1 described one of the main advantages of the new engine, the exchange of the robots internal states. This section points out the shortcomings of the old engine and shows how they are addressed in this implementation. In order to compare the old engine with the new one, the game play for a complete RoboCup Middle Size league game was converted to the new engine. The old engine consists of one *Globals* definition that can be compared with the Masterplan of the new engine. This *Globals* definition pointed to 8 *policy* files containing the contexts with the behaviours for standard situations and normal game play. In the new engine there are 20 plans with 85 states that represent this behaviour. At first this might look more complicated, but it is split up into smaller re-usable plans. See Scharf [23], for a more detailed explanation. The behaviours stayed almost the same with a few modifications, such as that they are now independent from the context in which they are executed.

The RoleAssignment of the old engine is split up into a Task- and RoleAssignment. Allocation of tasks occurs whenever a robot enters a state containing a plan and periodically for

each plan in execution. At the initial execution of the plan, the robot commits to a task. During the execution of that plan the periodically check evaluates if the plan can achieve a higher utility value by changing the task allocation. This is related to the RoleAssignment that was executed as a behaviour in the old engine, but with the main advantage of a non oscillating task allocation. This is because of a utility threshold on each plan and that the allocation of tasks is part of the engine and not programmed into a "special behaviour". Table 6.1 points out the main differences between the old and the new engine.

	Old Engine	**New Engine**
Hierachies	2	n
TeamPlay	not supported	supported
RoleAssignment	as a behaviour hack	supported
Explicit modeling of Conditions	just inside behaviours	supported on plans

Table 6.1: Main Differences between the Old and New Engine

Changing the task on a plan is shown in Figure 6.2 and referred to as *Utility Change*. The figure is an extract out of the complete Kick-off situation with the following game play in Appendix B. The test was done with 6 robots, but Appendix B shows only 4, because showing the plan trees of all 6 robots will exceed the scope of the appendix. Robot *Muecke* is closer to the ball and becomes the new attacker. Robot *Scotti* notices that another robot has taken the AttackTask and commits to the DefendTask. The third robot, *Zwerg* also calculates the utility and knows that another robot should take the AttackTask, marked in the figure as "Other Utility Change". *Zwerg* and *Scotti* notice the utility change simultaneously. The clocks of the robots have been synchronised for these tests, but this is only necessary to obtain debug data as shown in Figure 6.2. Traversing the plan tree and adjusting the assignments is triggered every $30ms$. Events within these $30ms$ are deemed to be simultaneously and shown as one time slot.

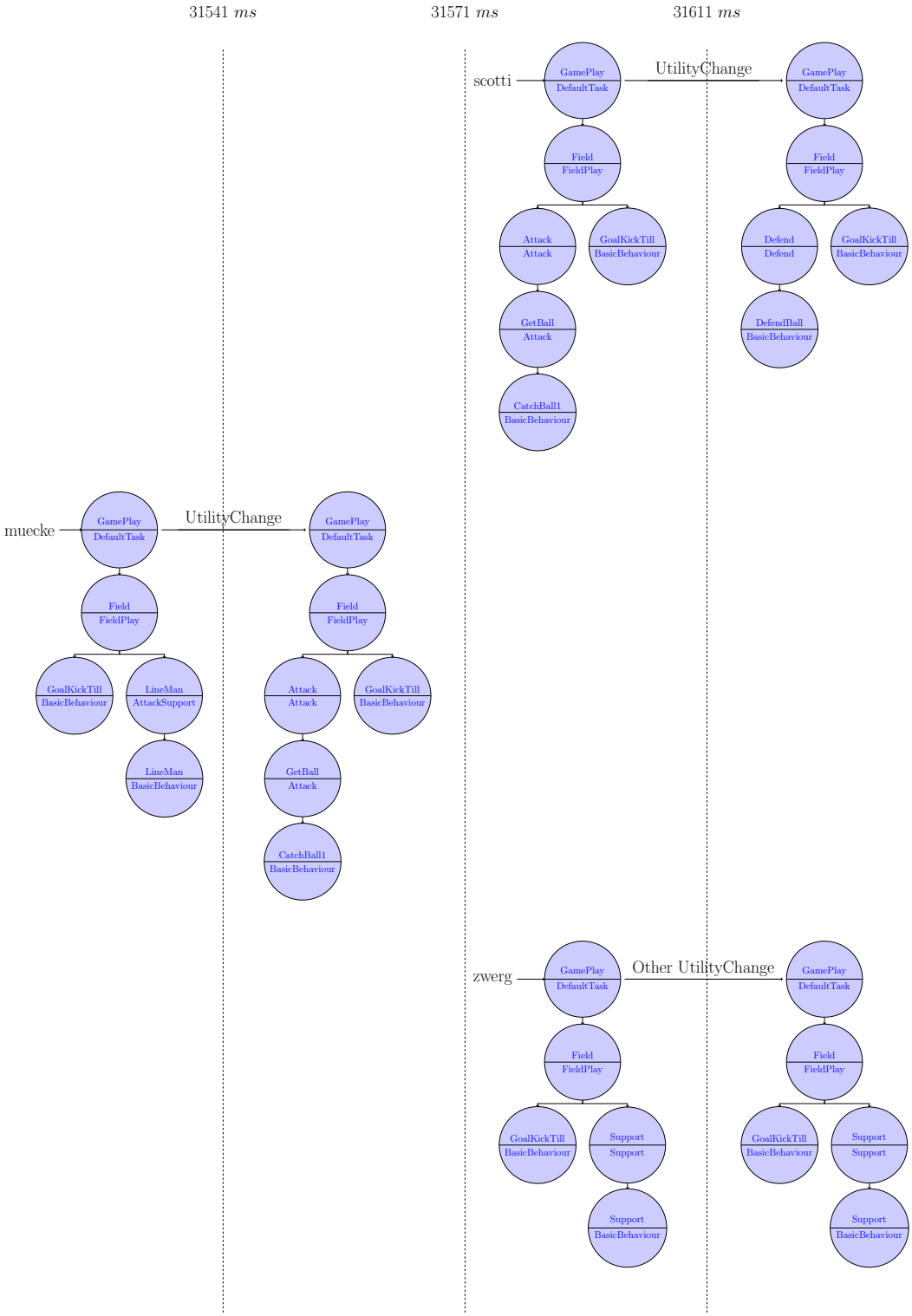


Figure 6.2: Utility Change: Two Robots swap their Tasks

The RoleAssignment in the new engine assigns roles to robots according to their capabilities and abilities and is just triggered if a robot enters or leaves the team.

The special *Observer* behaviour of the old engine which monitors certain fields inside the robot's world model and reacts on them, is now replaced. Inside the new engine this logical part is encapsulated into conditions on transitions.

Section 3.5 describes how team play is realised in the old engine. It cannot be seen as real team play because the robots do not know what their team members are doing. The *Timer behaviour* is responsible to wait for a certain amount of time and assume that the team member has finished its task. Timers are build-in in the new engine too, but they are just needed in failure cases. Figure 6.3 shows how a Kick-off situation is modelled in the new engine without using a wait timer.

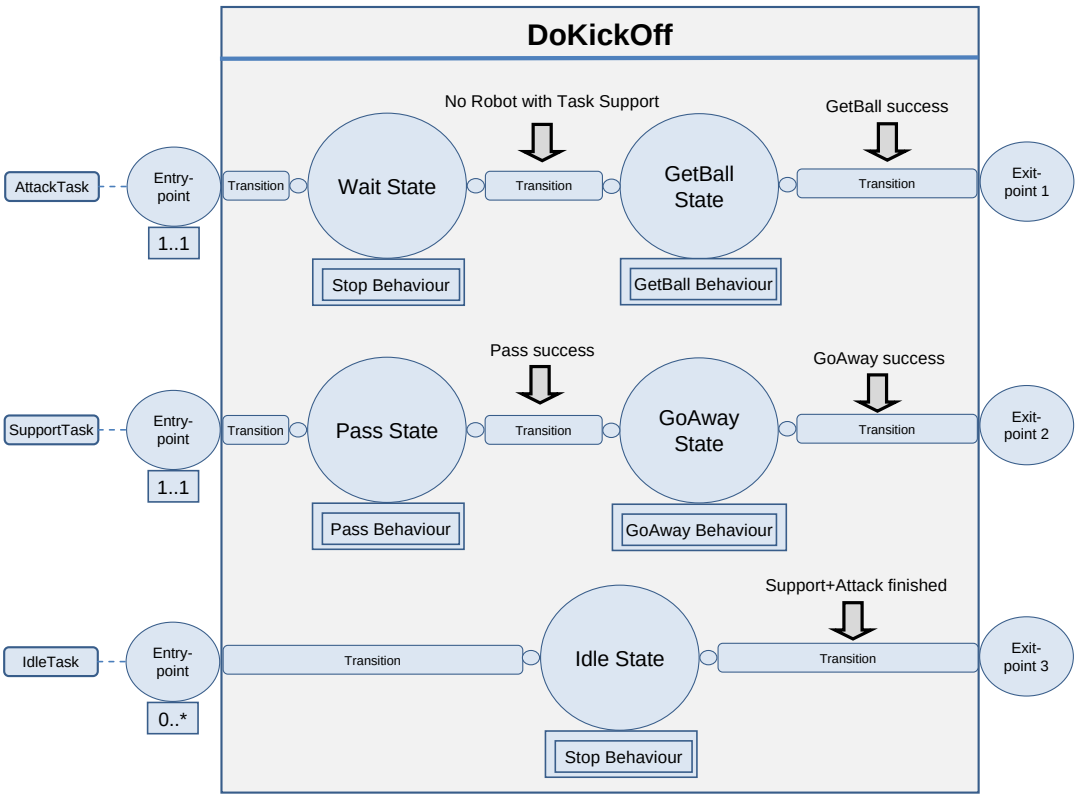


Figure 6.3: Plan to Play a Kick-off with 2 or More Robots

The plan *DoKickOff* needs at least two robots, one committing to the *AttackTask* and one to the *SupportTask*. The *IdleTask* is optional because there are only two robots needed to

execute the Kick-off. Basically the plan specifies the following: The support robot passes the ball to the attacker and makes room for the attacker to attack. The attack robot enters the *Wait State* where it monitors the transition to the *GetBall State*. In the old engine there is a timer at this place, but because of the exchanged plan trees in the new engine, the attacker waits for the supporter to finish its task. If the supporter leaves the plan, the attacker gets the ball and leaves the *DoKickOff* plan, too.

6.3 Performance Results

This section provides some runtime measurements. The time for a robot to traverse its plan tree to check all conditions and utilities including the change of that tree is measured. Most of the time the robot only needs to check the conditions and utilities which is usually below $2ms$. Peaks in this value occur if the plan tree needs to be changed due to a triggered transition or a utility change for example. The average time, together with the amount of peaks is shown for different situations in Table 6.2. Peaks are shown as average peaks per robot in the given situation.

Situation	**Avg. RT**	**Peaks** $> 10ms$	**Peaks** $> 50ms$	**Peaks** $> 100ms$
Kickoff	$1.2198ms$	1.18	0.479	0.26
Kickoff-Opponent	$1.3592ms$	3.6	1.14	0.98
Free Kick	$1.2808ms$	3.19	0.32	0.2
Free Kick-Opponent	$1.1875ms$	3.73	1.288	1.0
Throw-In	$1.0816ms$	3.26	0.04	0.0
3x10min Game play	$1.926ms$	99	7.6	3.06

Table 6.2: Overview about Average Times to Change the Plan Tree

Runtime peaks shown in Table 6.2 are the result of slow data structures used in the implementation to evaluate utility functions while creating the *RunningPlans* for all plans and behaviours within a state. Future work will improve that.

The positioning in the evaluated standard situations shown in Table 6.2 are deterministic. This means, given the same robots with the same roles, results in the same positioning for that situation. In the DoKickOff example mentioned above, the supporter role always takes the position of the pass player and the attacker role the position of the pass receiver. The reason for this deterministic positioning is because of the “priority - task mapping”. The

decision which position to take is only based on priorities. If the decision is also based on uncertain environment data, it would not be deterministic.

To show how well the robots play together, Section 6.4 introduces a measurement that is able to reflect that.

6.4 TeamPlay Results

The most interesting plan of the modelled game play behaviour is the *GamePlan*, shown in Figure 6.4.

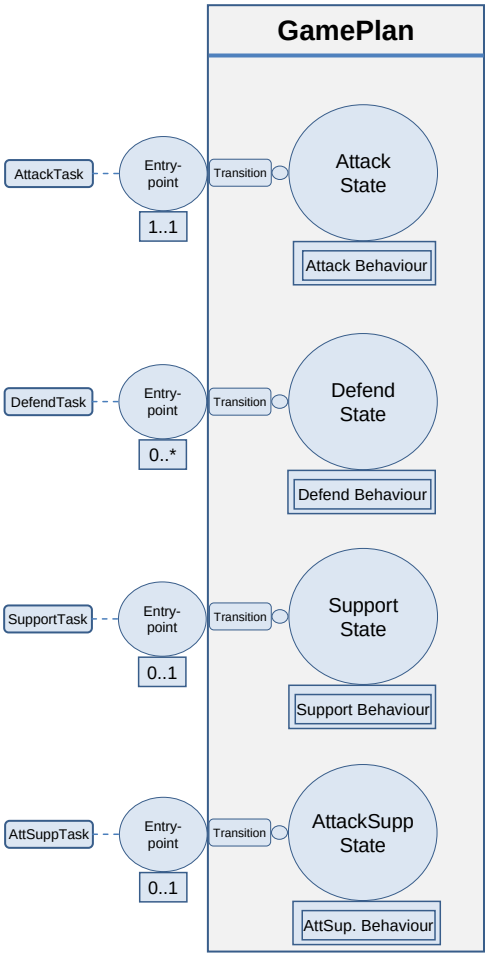


Figure 6.4: The GamePlan that Describes the Game Play Behaviour.

It consists of four entrypoints annotated with the tasks: Attack, Support, Defend, and AttackSupport. The goal keeper selects a different plan on a higher level, so it is not playing the *GamePlan* with the others. All tasks, except for Defend have a maximum cardinality of 1. Task Attack is the only one with a minimum cardinality, there needs to be exactly one attacker. The decision on which entrypoint to select for each robot is not only based on the priorities of the roles towards the according tasks, but also on two other summands in the utility function. One summand expresses the distance to the shared ball (the ball, two or more robots agree on) and another summand expresses the distance to the own goal. Every robot calculates the following utility function:

$$\begin{aligned} Utility \quad = \quad & w_1 * \sum_{t \in tasks} \sum_{x \in t} \left(\frac{Prio\,(role_x, t)}{\#robots} \right) + w_2 * \max_{y \in Attack} \left(1 - \frac{dist\,(y, sharedBall)}{fieldDiag} \right) + \\ & w_3 * \max_{z \in Defend} \left(1 - \frac{dist\,(z, ownGoal)}{fieldDiag} \right) \end{aligned}$$

This utility function returns the maximum value, if the closest robot to the shared ball commits to the Attack task and the closest to the own goal commits to the Defend task. The priority summand acts as a basic tendency for the robot towards a task. In this evaluation the following values are used: $w_1 = 0.15$, $w_2 = 1.0$, $w_3 = 0.01$. Priorities of a role towards a task are set to 0.6 if the robot should have a preference towards a task, otherwise it is set to 0.5.

One way to measure how well the robots play together is based on the length of time spans a team of robots does not believe in the same assignment. A perfect situation is, if all robots believe in the same assignment. This is because either every robot calculated the same assignment, or corrected it while receiving plan trees from team members. Because of the ongoing utility checks, a change in the assignment can occur if a new assignment matches the situation better than the current one.

Whenever a robot calculates a new assignment such that it needs to change its task, it enters another state and thus executes other plans or behaviours, resulting in a new plan tree. Other robots receive this plan tree and if they did not calculate the same new assignment already, update their current assignment according to it. The best case is that all robots notice that a change of the assignment is necessary and adapt to the new assignment simultaneously.

Until all robots are aware of the new assignment some time will pass, because they communicate their plan trees on a pre-defined frequency and not event based. In addition the network might add some latency. The average time between the first robot decides to change the assignment, until all team members agree on an assignment again is shown as one measurement unit in the following diagrams.

The tests for the following diagrams were done as follows: At first, the engine on all participating robots was started. After they had exchanged their plan trees[1], a Kick-off command was send by the *Referee Box* tool used in the RoboCup Middle Size league. Once they finished positioning for the Kick-off, the start signal was send. Each test lasted for 5 minutes with a simulated half time in the middle. Half time was simulated as sending the stop signal and then a new Kick-off signal, again followed by a start signal. In simulator tests the ball position was reset to the middle of the field after each goal. In the tests with the real robots the ball was taken away from the attacking robot shortly before the goal and kicked to one of its team members to enforce a utility change.

For every data point in the packet loss and packet delay diagrams, one of these tests was performed. One test is an episode of 5 minutes containing about 7.77 utility changes per minute. The average times for the robots to agree on an assignment is used as a data point in the diagrams.

Figure 6.5 illustrates a game in the simulator with 6 robots, lasting 5 minutes. The communication is realised via wireless network. No packet loss occurred during this test and the network delay was about $10ms$. One robot, the goal keeper is not playing the *GamePlan* so the maximum of different assignments is 5. There are 2 Kick-off situations in the graph, one at the beginning and one at half time. In these situations the graph shows that all robots agree on one assignment. This is because plans other than the *GamePlan* are not taken into account in this and the following diagrams. The reason for not taking other plans into account is that their utility depends only on the priority summand, not on the situation.

[1]The monitoring tool *LebtClient* shows that.

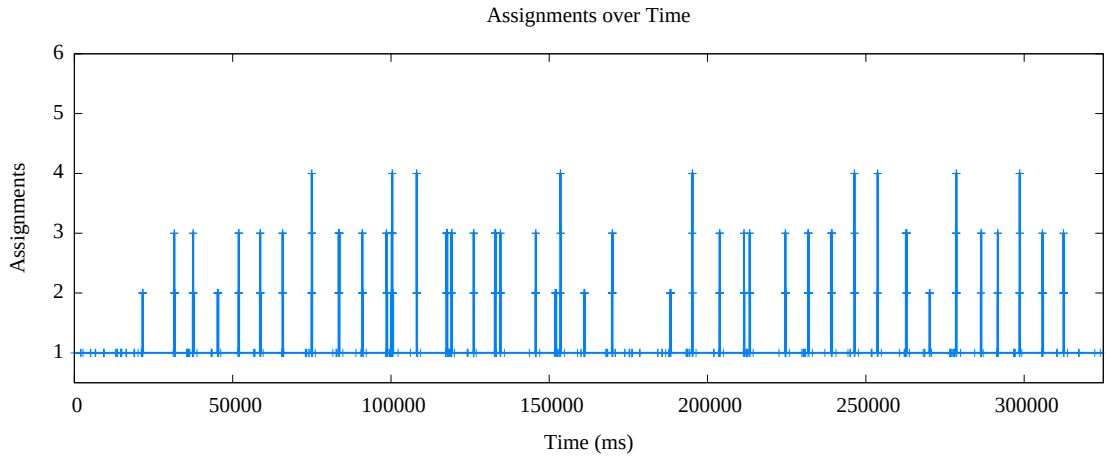


Figure 6.5: Simulator: 6 Robots in a 5min Game with 2 Kickoffs (Start and Half Time)

The number of different assignments over time is shown in Figure 6.5. Most of the time the robots agree on one assignment, but there are peaks in between where they do not agree. Peaks up to 3 different assignments, mainly happen because of a utility change. Some robots still have the old assignment, some have the new assignment already, and others have a mixture of both. The average time the robots need to agree on a new assignment is $177ms$ with a deviation of $89ms$. Statements about the frequency of peaks cannot be made, because it depends on the situation if a utility change occurs. It seems to be regular in Figure 6.5, but this is because the test should be comparable and thus the simulator reset the ball to the middle of the field after each goal, resulting in almost the same time difference between the peaks.

Figure 6.6 zooms into the fifth peak of Figure 6.5.

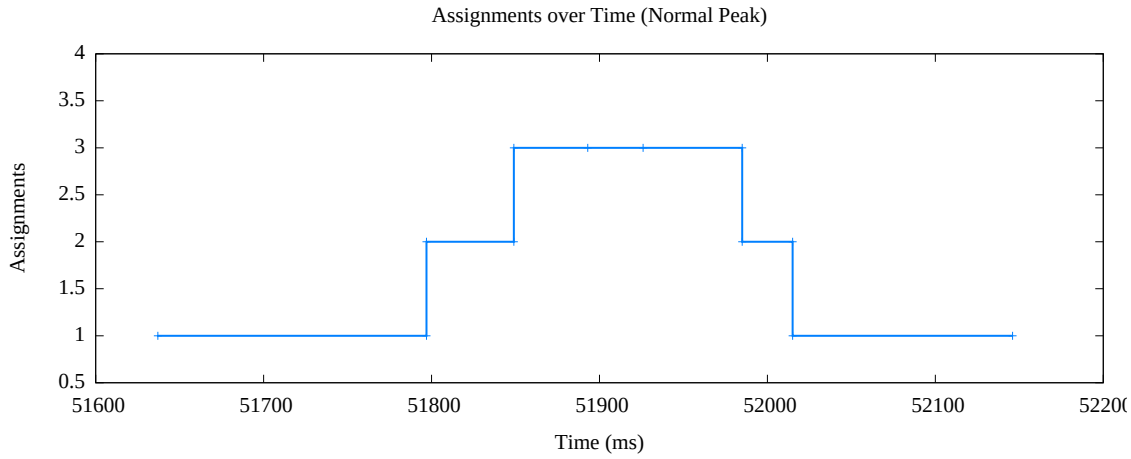


Figure 6.6: Simulator: One Peak in 6 Robots Playing a 5min Game

Before the peak the assignment contained the following robots with their tasks:

AttackSupport None

Support Zwerg

Defend Bart, Fransen, Scotti

Attack Muecke

The peak lasted for 218ms and started with the robots Bart and Zwerg deciding simultaneously that a utility change should be performed. Due to this change, Bart should become the new Attacker and Muecke the new AttackSupporter. 52ms later Zwerg receives an old plan tree from Bart telling that Bart is still a Defender, resulting in 3 different assignments at that time. Again 44ms later the robots Fransen and Scotti notice that a utility change should be performed with the same result as Bart and Zwerg calculated before. On the next event, 33ms later, Zwerg corrects its assumption and has the correct new assignment as Bart again. But at the same time Fransen and Scotti receive the plan tree from Muecke that it is still the Attacker so their assignment is invalid because of 2 Attackers. 59ms later Muecke notices that a utility change is needed and adapts to the new correct assignment. Within the same time slot, Fransen receives the new plan tree from Muecke and believes that Bart is the only new Attacker. Finally Scotti is the only robot that has a wrong assignment, but updates it 30ms later, too.

A second measurement unit to determine how well the robots play together is the average

amount of different belief states clustered wrt. the allocation within the *GamePlan*. It expresses the average amount of different believed assignments. In Figure 6.5, the average belief count is 1.028 with a deviation of 0.216. It cannot be exactly 1, because of the utility changes, leading to different assignments until the new assignment is believed by all team members.

In the next section the presented measurement units are used to evaluate tests with packet loss and packet delay.

6.5 Simulator Tests with Packet Loss and Delay

Simulation of packet loss is realised as *iptables*[2] rules on every robot. Iptables is a packet filter concept for UNIX systems. Rules are set up to discard incoming packets from other robots with a given probability. Packet delay is simulated by an iptables rule that puts all packets received from other robots into a queue. A small program[3], written in Perl fetches the packets from that queue and delays them by a given time. The script was enhanced to use pseudo random delays between a lower and upper bound in milliseconds. Packets, which are outgoing to other robots, are not blocked at all.

Figure 6.7 shows how long the robots need to agree on a new assignment, if packets are dropped. Up to 70 percent loss there is only a slow rise in the average time. Beyond this, the team play starts to suffer dramatically. Up to 40 percent the time to agree hardly reaches $300ms$, after that the curve becomes steeper with $370ms$, $496ms$ and $844ms$ at 50, 60, 70 percent, respectively. A jump occurs at 80 percent, where the average time is $3894ms$.

[2] http://www.netfilter.org last accessed 12/25/2008
[3] http://people.redhat.com/berrange/notes/network-delay/delay-net.pl last accessed 12/25/2008

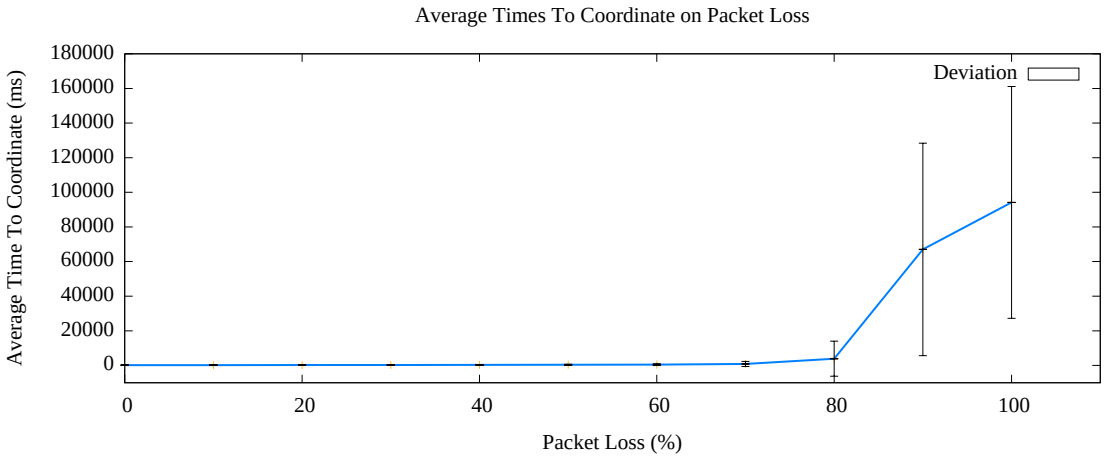


Figure 6.7: Simulator: Average Time to Coordinate with Packet Loss

Figure 6.8 zooms on 0 to 70 percent of Figure 6.7 and shows the deviation of the values more clearly.

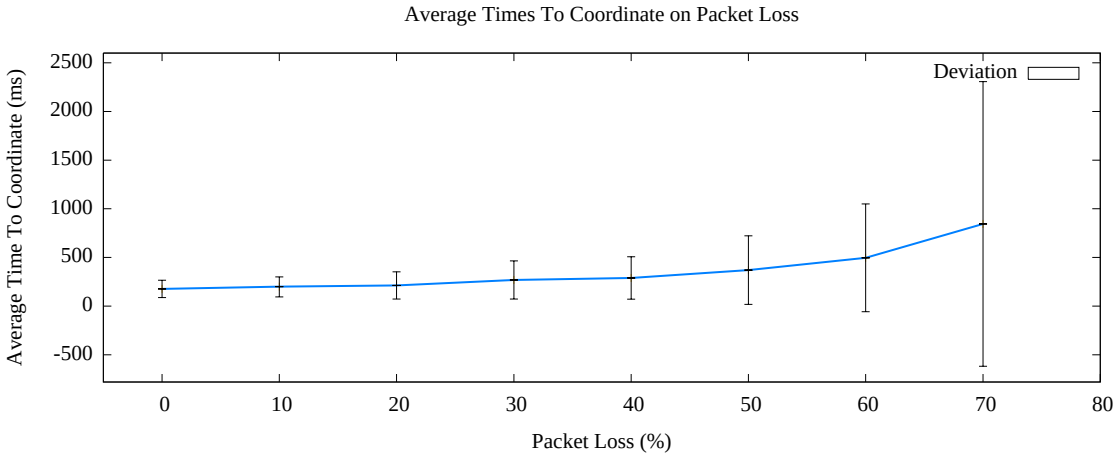


Figure 6.8: Simulator: Average Time to Coordinate with Packet Loss (zoomed)

The standard deviation in Figures 6.7, 6.8, as well as in Figures 6.12, 6.14, and 6.16, increases continuously. In general the time, the robots need to agree on an assignment, increases during packet loss and packet delay. This is because it takes longer to receive the information that reflects the current situation. If packets are lost, a robot sticks to the current assignment until it is able to receive new data. The frequency of received data has a higher jitter and

so does the time to agree on an assignment.

If packets are delayed, the robot continuously receives data, but this data does not reflect the current situation, it reflects one in the past. A robot that has calculated a new assignment due to a better utility value in the current situation, might overwrite it with data from the past. If the robot updates the assignment because of these old data, but still believes that there would be an assignment with a better utility value, it needs to re-update the assignment with what it believes. This continues until the old information from team members reflect the same assignment.

The higher the packet loss or delay, the higher the probability that an agreement on a new assignment is just a coincidence of all robots receiving data that reflects the current situation within a short time span. The diagram in Figure 6.9 shows that small peaks are rare. Diagrams for packet loss below 80% and for packet delay are similar, but contain more such small peaks. A peak that does not last for a long time emerges if all robots receive data from their team members almost simultaneously. Because packet loss was simulated by setting a probability to drop packets, some robots might receive data from their team members at a given time, but others discard several incoming data consecutively. Such a situation splits the team into robots believing what their team members are doing and those who do not know about their team members current actions. The deviation increases because in some rare cases all robots receive data almost simultaneously, so they agree a lot faster than the average time

Because of the increasing deviation, using the average time to agree on a new assignment as a measurement unit does not lead to precise results. Nevertheless, it is a rough indicator of how well the robots play together.

The peaks in Figure 6.9 last a lot longer than in Figure 6.5 and they also overlap quite often. This is because messages are dropped, so the robots are almost unable to agree on a new assignment before the next utility change occurs. In the middle where all the robots agree is the simulated half time where all robots were stopped and executed the *Kickoff* plan, whose assignment is not taken into account in these graphs as mentioned before.

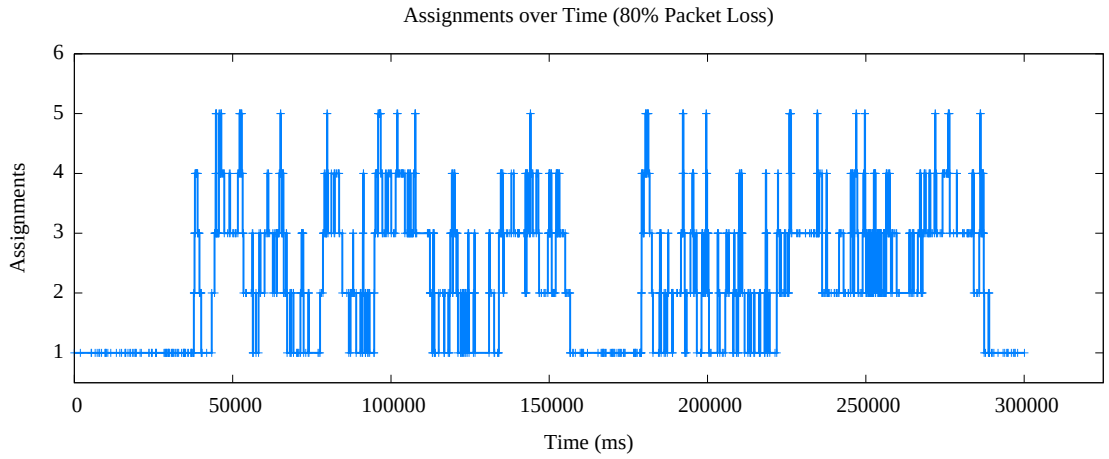


Figure 6.9: Simulator: Average Time to Coordinate with 80 Percent Packet Loss

Figure 6.10 zooms at the second peak of Figure 6.9. This peak is a lot longer that the one described before and lasted for $12974ms$.

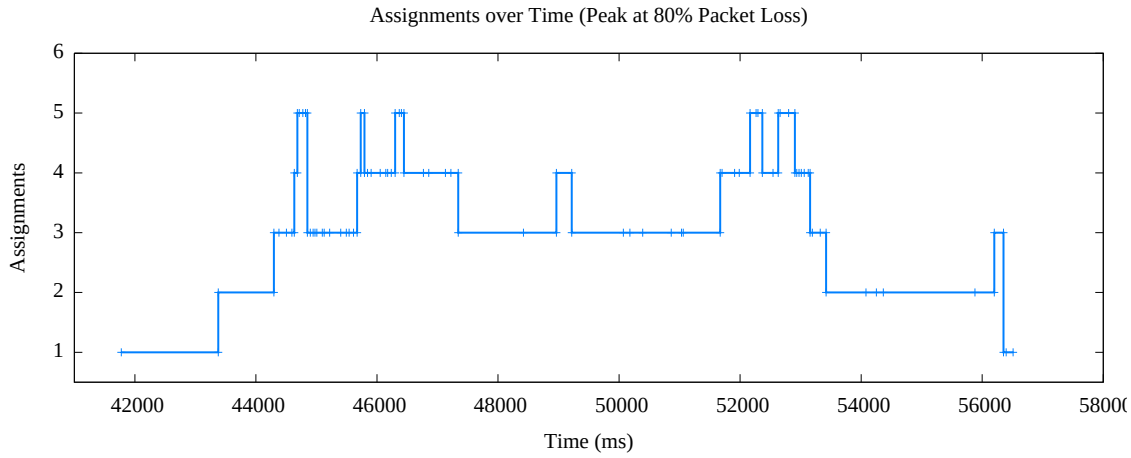


Figure 6.10: Simulator: One Peak in Negotiating Times with 80 Percent Packet Loss

At the beginning of this peak the following assignment existed:

AttackSupport Muecke

Support None

Defend Bart, Fransen, Scotti

Attack Zwerg

Explaining all the details within this peak would exceed the scope of this section, but what basically happens within this peak is that a utility change occurs and all robots do not agree on the same assignment before the next utility change occurs. There are 5 utility changes within this peak and the robots have problems receiving the assignments from their team members. Some plan tree messages from one utility change even reach their recipients in the middle of the next utility change leading to confusion. Figure 6.9 also shows that the robots hardly agree on just one assignment.

The diagram in Figure 6.11 shows the amount of average belief states with packet loss. In average there are 7.77 utility changes per minute. The more packet loss, the more different assignments exist. With no communication, all robots believe they play alone and thus they need to satisfy the cardinalities on the *GamePlan*, that enforces at least one robot committing to the AttackTask. That is why there are almost 5 different assignments. The reason for not exactly being 5 assignments is that the values take the situations after the Kick-offs into account where not all robots join the *GamePlan* simultaneously. At first the robot with task support joins the *GamePlan*, after he played the pass to the robot performing the attack task, and calculates an assignment. The other robots are still in the *Kickoff-Plan* and follow later. While the robots are in the *Kickoff-Plan*, they do not have an assignment for the *GamePlan*.

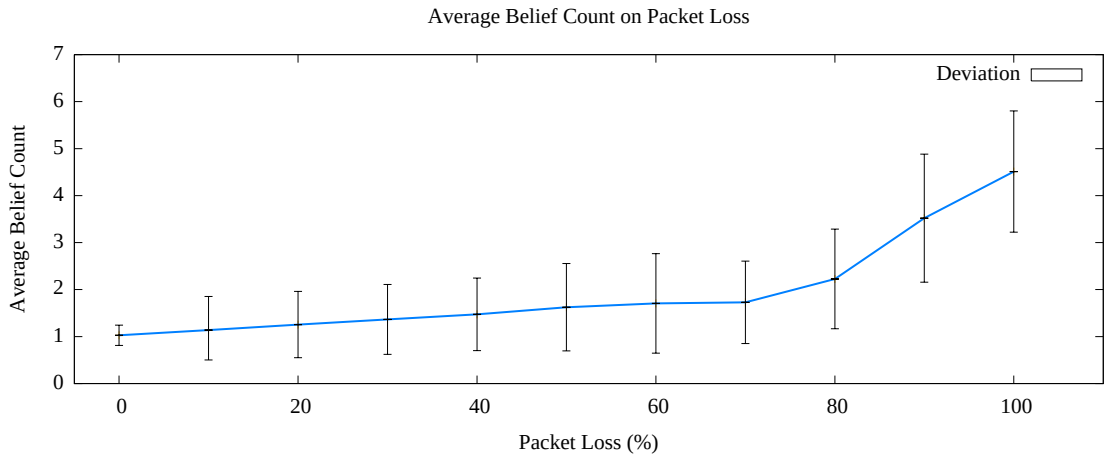


Figure 6.11: Simulator: Average Count of Belief States with Packet Loss

Tests with packet delay are shown in Figures 6.12 and 6.13. x-values in graphs that deal with

packet delay represent the middle between the upper and lower bound, set for generating packet delay. For example the value 50 results from pseudo random packet delay between 25 and 75ms. Other values are also shown as the middle of a 50ms time span. Figure 6.12 illustrates the average time a team of 6 robots needs to agree on a new assignment, if the communication between them is delayed. 50ms delay already results in an average time of 263ms to agree on a new assignment. It continues to rise slowly. With a delay around 100ms it is already as bad as having a packet loss of 40 percent. At 500ms delay it takes about 499ms for the robots to agree, in average. Finally, with a delay of 1300ms they need 747ms. In Figure 6.13, the average amount of belief states with delayed communication is shown.

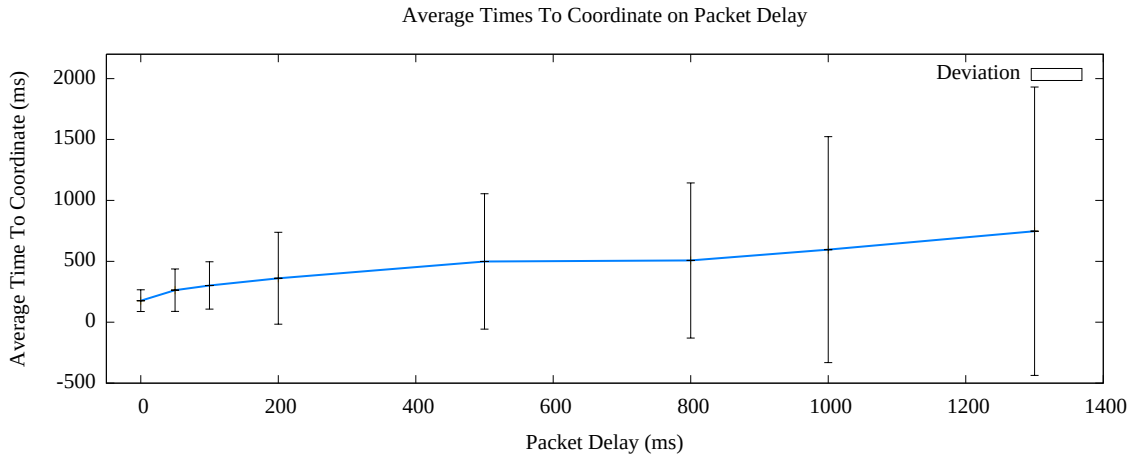


Figure 6.12: Simulator: Average Time to Coordinate with Packet Delay

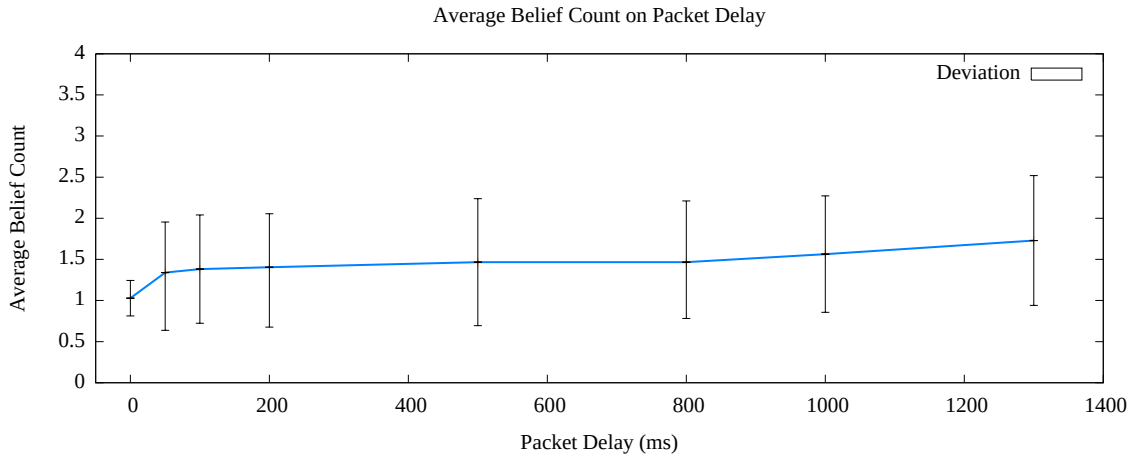


Figure 6.13: Simulator: Average Count of Belief States with Packet Delay

Surprisingly, the amount of belief states on packet delay up to $1300ms$ is relatively stable between 1 and 1.5 states. This is probably because there are not that many different assignments due to lost packets, but old packets and thus assuming old assignments are still valid rather than creating assignments that represent a mix of the old and new assignment on a utility change.

In general one can say that packet delay has an even worse effect on team play than packet loss has. Times to achieve the same belief keep increasing because every time a robot tries to adapt itself to the new situation, it receives old information from its team members. This old information leads to updates on the assignments of the plans the robot is executing. For instance, if RobotA supposes that RobotB also joins PlanA, RobotB is integrated into the assignment of PlanA. RobotB may execute PlanA too, but now the delayed packets arrive at RobotA and it removes RobotB from the assignment until the new packets, containing information that RobotB actually executes PlanA, arrive. A loss of packets can be compensated until a certain level, because a robot does not remove a team member from any assignment if it does not receive anything from it for a certain time span. During this time span[4], the robot sticks to the assignment. After the time span, it removes the robot, of whom it does not receive any data, from the team and thus from the assignments.

[4]Currently set to 2 seconds.

6.6 Tests on Real Middle Size Robots with Packet Loss and Delay

Tests on real Middle Size robots of the CarpeNoctem RoboCup team were done with just four robots instead of six as in the simulator. That is why the results are not directly comparable. A smaller team of robots should come to an agreement faster than a larger one.

Figure 6.14 shows how long the real robots needed to agree on a new assignment with packet loss.

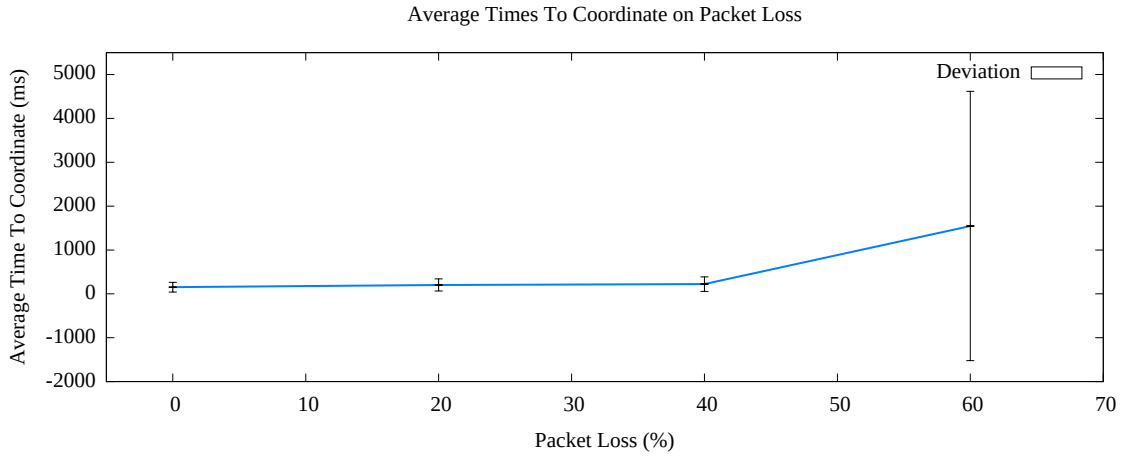


Figure 6.14: Real Robots: Average Time to Coordinate with Packet Loss

Times are similar to the ones measured in simulator tests. Up to 40 percent packet loss, the average time to agree on a new assignment is around $200ms$. The average time to agree starts to increase significantly at 60 percent.

The diagram in Figure 6.15 illustrates the average count of belief states in the team while experiencing packet loss.

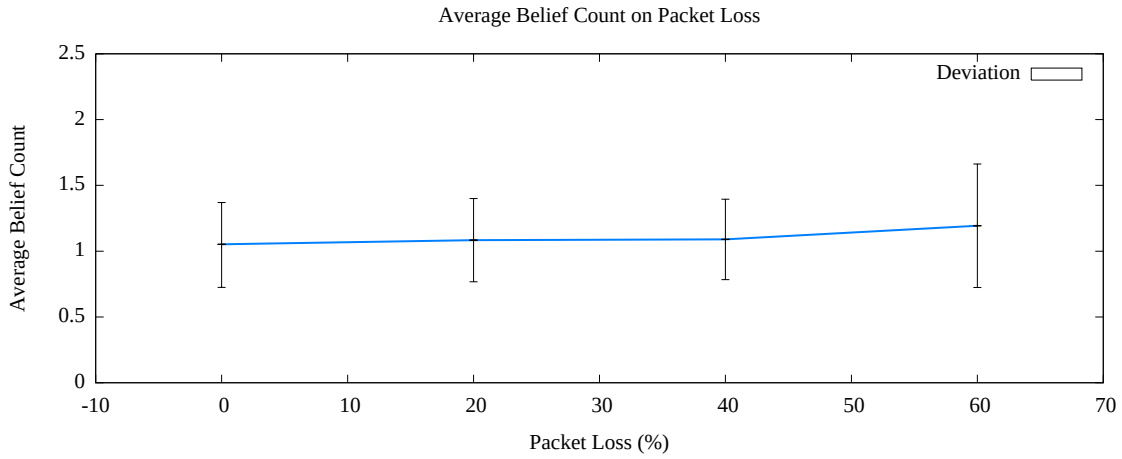


Figure 6.15: Real Robots: Average Count of Belief States with Packet Loss

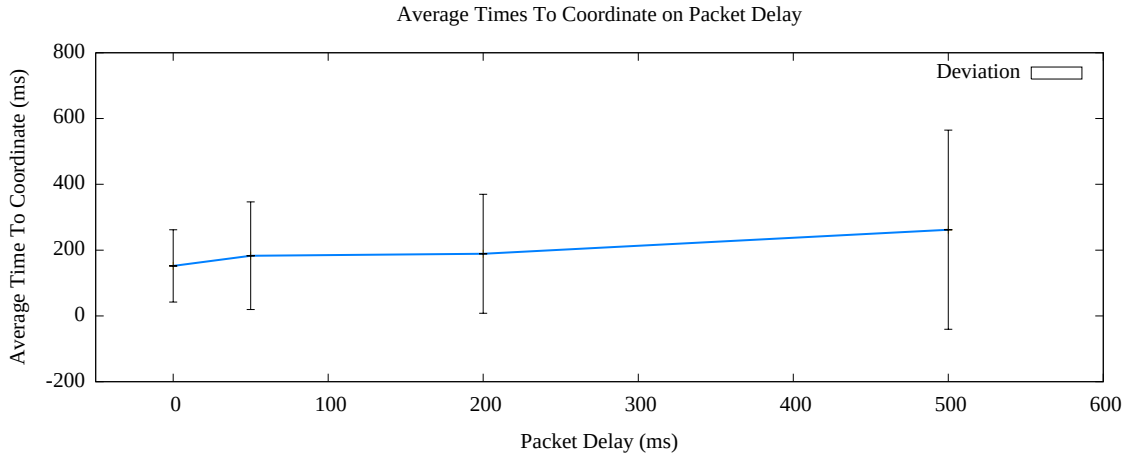


Figure 6.16: Real Robots: Average Time to Coordinate with Packet Delay

Surprisingly, the delay turned out as not effecting team play as much as in the simulator. This is probably because real tests were only done with four instead of 6 robots, so there were less contradictory old plan trees that led to updates on the assignments again.

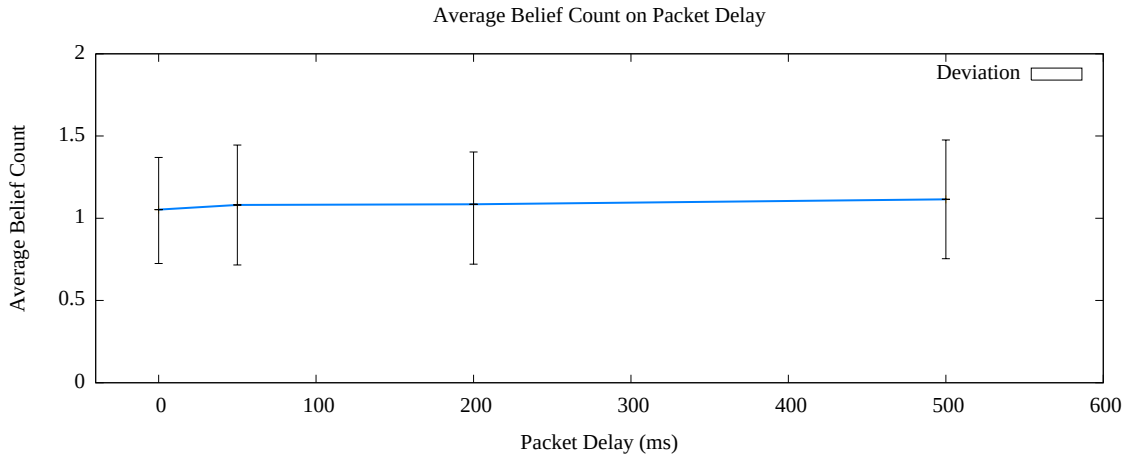


Figure 6.17: Real Robots: Average Count of Belief States with Packet Delay

Figure 6.17 supports the assumption stated before that the reason for faster agreement times is because of less robots. In average there are less belief states than in the simulator tests.

6.7 Other Results

During the evaluation some bugs were found and corrected. One important fact that turned out while using the world model of the old engine is that the world model was designed to calculate requested data on demand instead of caching it. That was not a problem, because the data was usually requested by behaviours only, with a frequency of $30Hz$. The new engine requests data a lot more often and so the world model had to be changed to handle it, because otherwise there would be very high peaks in the execution times of the engine. The new engine accesses the world model, e.g., when it needs to calculate an assignment for a plan. While searching for an assignment, it takes data from the world model into account, so it calls the functions inside the world model more often.

Some tests showed that the assignment of tasks to robots by just using the priorities for tasks, defined in the roles of the robots, were not deterministic. This was because of the inaccuracy of the *double* value used on comparing utilities with each other. This was solved by restricting it to certain accuracy.

Further tests showed that a robot's calculated assignment was updated not just by com-

paring it to newly received plan trees from team members, but also on older, cached plan trees. This resulted in side effects that a robot did not wait properly before another one had finished its task. The *DoKickOff* plan, shown in Figure 6.3 includes such a waiting condition where the robot did not wait properly.

The evaluation in this chapter showed how the engine implemented with this thesis performs under certain circumstances. In the next section a summary of the thesis is presented as well as possible future work that can be done to improve the engine.

7 Summary and Future Work

The final chapter of this thesis sums up what was done in this thesis and outlines future work that can be done to enhance the BehaviourEngine implemented for this thesis.

7.1 Summary

In the area of autonomous mobile robots there are several software problems to solve, besides the hardware problems such as building sensors and actors. One of the software problems is to control the actions of a robot. This problem can be addressed by machine learning so the robot learns how to control itself, or a human specifies how the robots behave in certain situations. The problem of specifying the behaviour of multi-agent systems is that it is complex and not easy to do in a whole. That is why the complex behaviour needs to be split up into smaller pieces. Each piece can then solve a particular problem and later be combined to achieve various complex behaviours. This thesis addresses the problem of executing these pieces as specified by the user, but leaving the robot room to make its own autonomous decisions.

The foundation for the implementation presented in this thesis is ALICA – A Language for Interacting Cooperative Agents. ALICA specifies how robots use plans to solve problems as a team. This thesis creates the first implementation of this language and will be used as the BehaviourEngine to control the robots of the CarpeNoctem Middle Size RoboCup team.

The BehaviourEngine is written using the programming language C# to suit into the CarpeNoctem software framework. It is programmed in a modular way with specified interfaces, so the modules can be easily replaced. Shortcomings of the former BehaviourEngine are taken into account and removed. The biggest improvement is the realisation of team

play. Team members do not only exchange parts of their world models but also their current internal state. This internal state represents all plans a robot is executing. Now it is possible for one robot to know what its team members are doing and take this into account when making decisions. Coordination, such as waiting for a team member to finish its task or changing to a certain state can easily be done. Such conditions are not strings that need to be parsed at runtime, but auto-generated source code, which is executed to evaluate the condition. This is done due to performance reasons.

Inside the modelled plans from a user, there are also autonomous decisions for a robot to take. For instance, if a robot starts to execute a plan that contains two or more sub-tasks it needs to decide for one of them. Once it has done this decision it is evaluated periodically. If the situation changes, so it might be better for two robots to swap their tasks, they will do it. The decision whether it is better to swap tasks or stay committed to their current ones is based on a utility function. Utility functions are also evaluated by executing auto-generated code. A second point where a robot can make an autonomous decision are plantypes. They represent a set of plans with the same result and the robot can choose one of them to achieve this result.

This BehaviourEngine was successfully tested in a simulator with six robots and with four real RoboCup Middle Size robots. Tests showed that this engine is robust against a certain amount of packet loss and packet delay, in a way that game play is only minimally negatively affected.

Some parts of the BehaviourEngine are still implemented as stubs and thus they could be subject to future work.

7.2 Future Work

This section points out which parts of the BehaviourEngine are implemented as stubs, so they need to be extended and gives an outline on where room for improvement is.

7.2.1 Conditions and Utilities

The most important task for future work will be the automatic code generation for conditions and utilities. At the moment the PlanDesigner generates empty functions that need

to be implemented by the user. A library for common conditions is included in the BehaviourEngine presented in this thesis. The user can choose from various functions to call within the generated stubs.

7.2.2 Plan Parameter

Behaviours can be parameterised to use them in different situations. Plans may have parameters too. The current implementation supports those parameters but is not capable to pass them down to sub-plans or behaviours, yet. ALICA specifies even more powerful parameters. They should be passed not only down to sub-plans, but also up to parent plans and back and forth in time for planning purposes. Conditions (Section 4.8) should not only be able to use parameters of the current plan, but also the parameters on parent plans. There needs to be a mechanism that allows forwarding of plan parameters to sub-plans.

7.2.3 Role Assignment

At the moment the assignment of roles is implemented as a static mapping of a robot to a role using a configuration file. But the RoleAssignment module registers on the robots world model to be notified if a robot joins or leaves the team. Future work can use this registration to trigger a reassignment of roles because the number of robots has changed. In addition, roles should be assigned according to a robots abilities and capabilities. This is very important in heterogenic teams where certain robots have special abilities and thus they should be coordinated based on them.

7.2.4 Synchronisation (Synctransitions)

Synchronisation as specified in ALICA is not implemented so far. In some cases, where a group of robots should solve a problem where they need to start simultaneously, an explicit synchronisation is necessary. Synchronisation can be realised through communication among the participating robots. Once they have achieved a mutual agreement on starting to solve the problem, they do so, simultaneously. In the current implementation, a group of robots is able to solve a problem with a plan together, but with no guarantee when every single robot starts executing its task.

7.2.5 Tracking of Robots through Plans

Evaluation tests showed that the BehaviourEngine is still working with a certain amount of packet loss or delay. If the communication becomes even worse the team behaviour suffers. One method to improve team play in such situations could be to track robots through plans. This means, if one robot receives the plan tree from a team member, it tries to predict what plans the team member will execute next.

Bibliography

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

[2] Joseph Bates. The role of emotion in believable agents. *Communications of the ACM*, 37:122–125, 1994.

[3] Michael Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

[4] Michael Bratman. What Is Intention? In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, pages 15–31. MIT Press, Cambridge, MA, 1990.

[5] Rodney A. Brooks. Intelligence without reason. pages 569–595. Morgan Kaufmann, 1991.

[6] Rodney A. Brooks. Intelligence without representation. Number 47 in Artificial Intelligence, pages 139–159. 1991.

[7] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artif. Intell.*, (2-3):213–261.

[8] Arie van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. URL: `http://homepages.cwi.nl/~arie/papers/dslbib.pdf` last accessed: 11/06/2008.

[9] Uwe Düffert, Matthias Jüngel, Tim Laue, Martin Lötzsch, Max Risler, and Thomas Röfer. GermanTeam 2002. In *RoboCup 2002 Robot Soccer World Cup*. Springer, 2002.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86:269–357, 1996.

[12] Barbara J. Grosz and Candace L Sidner. Plans for discourse. In *Intentions in Communication*. MIT Press, 1990.

[13] Martin Hitz and Gerti Kappel. *UML@Work: Von der Analyse zur Realisierung*. dpunkt-Verlag, Heidelberg, 1999. ISBN 3-932588-38-X.

[14] Jomi Fred Hübner, Jaime Simao Sichman, and Olivier Boissier. S-moise+: A middleware for developing organised multi-agent systems. In *International Workshop on Organizations in Multi-Agent Systems, from Organizations to Organization Oriented Programming in MAS (OOOP2005*, pages 64–78. Springer, 2006.

[15] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In *Guilherme Bittencourt and Geber L. Ramalho, editors, Proceedings of the 16th Brazilian Symposium on Artificial Intelligence (SBIA'02), LNAI 2507*, pages 118–128. Springer, 2002.

[16] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 177 (2):277–296, 2000.

[17] Jill Fain Lehman, John Laird, and Paul Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. MIT Press, 1996.

[18] Hector J. Levesque, Philip R. Cohen, and José H. T. Nunes. On Acting Together. In *Proc. of AAAI-90*, pages 94–99, Boston, MA, 1990.

[19] Martin Lötzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences), Lecture Notes in Artificial Intelligence*, pages 114–124. Springer, 2004.

[20] Hyacinth S. Nwana and Martlesham Heath. 1 To appear in: Knowledge Engineering Review, 1996 Software Agents: An Overview.

[21] Stephan Opfer. Efficient Decision Making in ALICA. Not published yet.

[22] Anand S. Rao and Michael P. Georgeff. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, 1995.

[23] Andreas Scharf. Grafische Verhaltensmodellierung kooperativer autonomer Robotersysteme, 2008.

[24] Nathan Schurr, Steven Okamoto, Rajiv T. Maheswaran, Paul Scerri, and Milind Tambe. Evolution of a teamwork model. In *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, pages 307–327. Cambridge University Press, 2005.

[25] Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993. ISSN 0004-3702.

[26] Jaime S. Sichmann. A social reasoning mechanism based on dependence networks. In *Proceedings the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 188–192, 1995.

[27] Hendrik Skubch, Michael Wagner, and Roland Reichle. A Language for Interactive Cooperative Agents. Technical report. Not published yet.

[28] Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[29] Michael Wooldridge. *An introduction to Multi-Agent Systems*. West Sussex, England: Joh Wiley & Sons Ltd., 2002.

[30] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10:115–152, 1995.

[31] Oliver Zweigle, Reinhard Lafrenz, Thorsten Buchheim, Uwe-Philipp Käppeler, Hamid Rajaie, Frank Schreiber, and Paul Levi. Cooperative Agent Behavior Based on Special Interaction Nets. In *IAS*, pages 651–659, 2006.

A Classdiagram

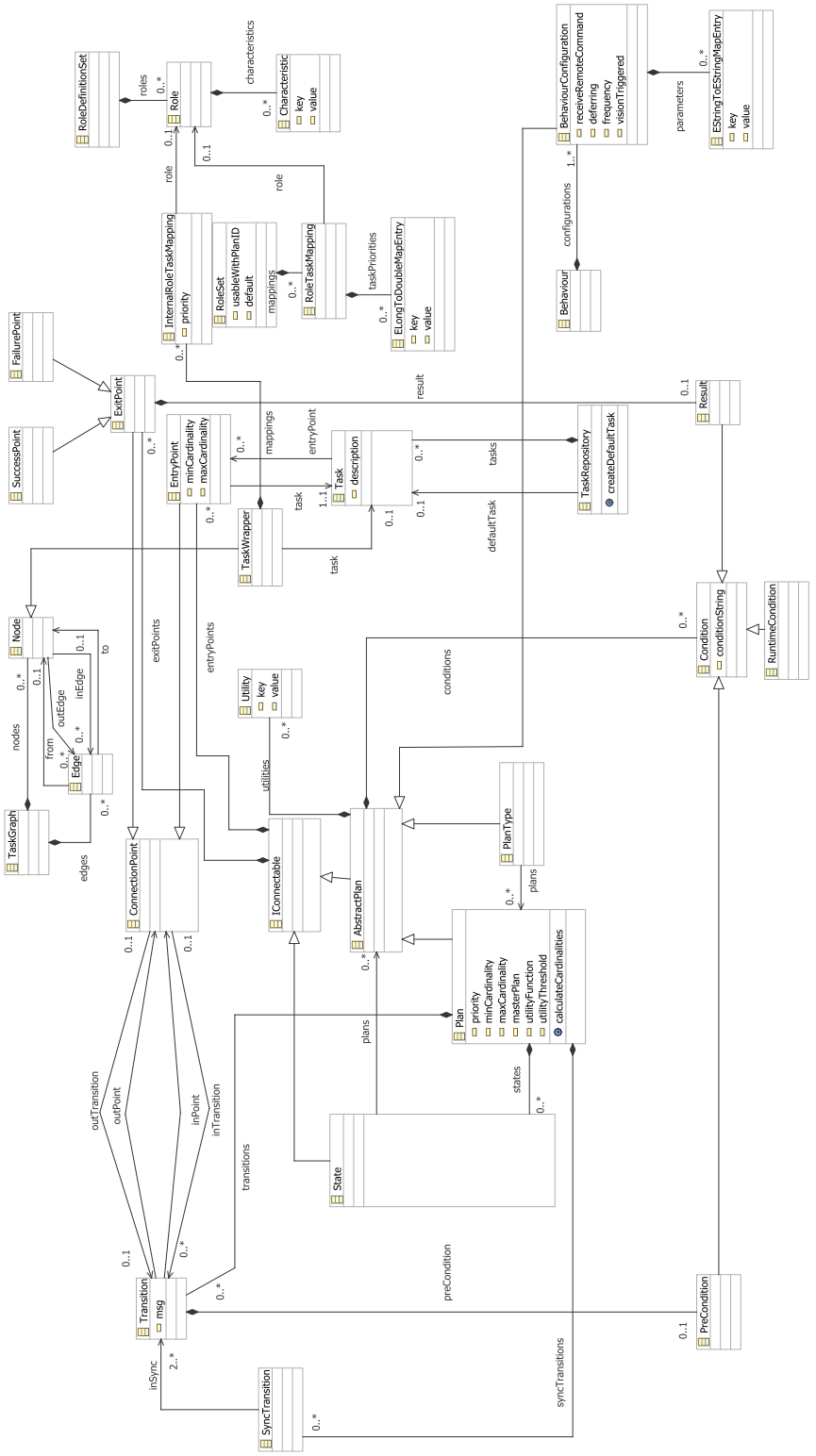


Figure A.1: Classdiagram

B Plantrees on Kick-off and Game Play