VERTEILTE SYSTEME

UNIKASSEL VERSITÄT

Plan Recognition and Tracking for Cooperative Autonomous Robots in Dynamic Environments

University of Kassel

Master Thesis of

Stefan Triller

At

Distributed Systems Group

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

Supervisor: Dipl.-Inf. Hendrik Skubch
Dipl.-Inf. Roland Reichle

March 31, 2010

Erklärung

Hiermit erkläre ich, dass ich die Masterarbeit 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 Masterarbeit habe ich bisher bei keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Baunatal, den 31.03.2010

Abstract

Cooperative multi-agent systems shall achieve a common goal and have to keep the team members informed about each other's actions. This can be achieved by direct communication or through recognition of changes in the environment. In a robotic scenario, both are either unreliable or imprecise, hence an agent cannot precisely know about the team member's current actions. This thesis targets on tracking the agents of a team, even if some information needed for tracking is unknown, instead of just relying on communication. The Dempster-Shafer theory of evidence is used to combine hypotheses about actions of an agent into hypotheses for a team of agents. It is also used to evaluate conditions which may lead to changes in an agent's behaviour. Results of this evaluation are used to rank team hypotheses, which describe the actions of a team, and thus to determine the most suitable one for the current situation.

Contents

1 Introduction **10**

2 Foundations **12**

2.1 Multi-Agent Systems 12

2.2 Dempster-Shafer Theory of Evidence 13

2.3 Predicate Logic 17

2.4 Bayesian Networks 20

3 Problem Definition **25**

3.1 Domain: RoboCup 25

3.2 Team Play 26

3.3 CarpeNoctem Behaviour Engine 27

3.4 Task Definition and Motivation 32

4 Related Work **34**

5 Approach **38**

5.1 Evaluation of Conditions 38

5.1.1 Expressing Lack of Knowledge 39

5.1.2 Implementation 40

5.1.2.1 Data Structures 41

5.1.2.2 Formula Evaluation 42

5.1.2.3 Eliminating Variables 47

5.2 Tracking Module 50

5.2.1 Team Hypotheses 52

5.2.1.1 Updating Team Hypotheses 54

5.2.1.2 Monitoring Conditions 56

5.2.1.3 Hypotheses Pruning . . . 57
5.2.1.4 Integrating WorldModel Information . . . 57
5.2.2 Synchronisation Points . . . 58
5.2.3 SharedBall Calculation Based on Dempster-Shafer . . . 60

6 Evaluation **62**
6.1 Representing Environment Data in the RoboCup Domain . . . 62
6.2 Scenario: Closest Robot to the Ball . . . 63
6.3 Handling Partly Known Information . . . 67
6.4 Handling Completely Unknown Information . . . 68
6.5 Hypotheses Ranking . . . 70

7 Summary and Future Work **73**
7.1 Summary . . . 73
7.2 Future Work . . . 74
7.2.1 Representing Alternatives in the Evaluator . . . 74
7.2.2 Implementation of a Tracking Module . . . 74
7.2.3 Detecting and Solving Team Play Conflicts . . . 75
7.2.4 Plan Recognition . . . 75
7.2.5 RoboCup: Opponent Tracking . . . 75

Bibliography **76**

List of Figures

2.1 Bayesian Network Example 21

3.1 ALICA Plan Example 29
3.2 CarpeNoctem Behaviour Engine 31

5.1 Bayesian Network Created by Variable Dependencies 48
5.2 CarpeNoctem Behaviour Engine with new Modules (red) 51

6.1 Closest Robot to the Ball with 2 Robots 63
6.2 2 Robots: Belief Vector, True, and False Part 65
6.3 5 Situations with 5 Robots 66
6.4 5 Robots: Belief Vector, True and False Part 67
6.5 Robot r_2 Monitoring if r_1 Has the Ball 68
6.6 Plan for Hypotheses Ranking 71

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, which 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.

In order to cooperate, the robots need to be aware of each others actions. The actions can be communicated explicitly within the team or inferred from changes in the environment. Unfortunately, both are subject to be unreliable and imprecise. Instead of only relying on communication or inferring actions from the environment, a robot can also track the actions of its team members.

This thesis targets at developing a tracking approach that is able to alleviate the effects of unreliable communication. A multi-agent system implementation (Triller [29]) based on the language ALICA (Skubch et al. [18]) that communicates the internal states of the team members is taken as a basis for this thesis. ALICA is based on plans that contain states in which agents execute behaviours or other plans. Agents change their states through transitions which are guarded by conditions. If a conditions is believed to hold, an agent *moves* along the corresponding transition. Plans have utility functions that indicate the most suitable combination of agents and parts of the plan. They also have pre- and runtime-conditions that need to hold in order to be executed. Conditions are expressed in predicate logic. These conditions and utilities are considered in a tracker approach within an evaluator for conditions that makes it possible for an agent to evaluate conditions in the view of its

team members. Since the agents of a team share a common plan library, such an evaluator offers the possibility to determine the states inhabited by team members. Because this information suffers from unreliability too, the Dempster-Shafer theory of Evidence (Shafer [15]) is used to express partial knowledge as well as total ignorance.

This work is organised as follows: Chapter 2 describes the foundations, which represent the background of this thesis. Chapter 3 explains the task of this thesis and introduces the problem domain. In Chapter 4, related work is presented that shows how others addressed the task of this thesis. The following chapter (Chapter 5) describes the approach done in this thesis to track agents in multi-agent systems. An evaluation of the approach is presented in Chapter 6, and Chapter 7 sums up the thesis and outlines future work.

2 Foundations

2.1 Multi-Agent Systems

A definition for the term *agent* in the context of computer software shall 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 [30]). He (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 sensor information again. Unfortunately, these sensor information are error prune and thus they are not 100% reliable.

In contrast to a single-agent system, 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.

An agent, in a multi-agent system, might depend on actions of other agents in order to achieve its goals. Sichmann [16] 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.

This section was a short introduction about software agents and multi-agent systems. Agents within a cooperative multi-agent system try to reach a common goal together. Unfortunately, if implemented on robots in real world environments, their sensor information may not be fully reliable. The next section presents a mathematical theory which can help to deal with this problem, the Dempster-Shafer theory of evidence.

2.2 Dempster-Shafer Theory of Evidence

The Dempster-Shafer Theory (DST) was invented in 1976 by Arthur Pentland Dempster and Glenn Shafer [15]. It extends traditional probability theory to express *total ignorance*. It is also known as the theory of belief functions. One good example to explain this is the weather forecast. Imagine the weather forecast predicts there is a 80% chance for sunshine tomorrow. But does that mean there is a 20% chance of rain? What about fog, a storm etc.? The answer is, the remaining 20% cannot be given to a single element, but to a group that contains rain, fog, storm, etc. One cannot make any assumption on how these 20% are distributed among the group members. This problem is addressed in the DST.

Smets [20] gives an overview about belief functions. Let Ω be a finite non-empty set called the **frame of discernment**. A mapping bel: $2^\Omega \to [0,1]$ is a **belief function** iff there exists a set of **basic belief assignments** (bba) m: $2^\Omega \to [0,1]$ such that:

$$m(\emptyset) = 0 \tag{2.1}$$

$$\sum_{A \subseteq \Omega} m(A) = 1 \tag{2.2}$$

and

$$bel(A) = \sum_{B \subseteq A, B \neq \emptyset} m(B). \tag{2.3}$$

The values of $m(A)$ for subsets A in Ω are referred to as **basic belief masses** (bbm). In contrast to traditional probability theory, bbms are assigned to any subset of Ω. A subset A of Ω, $|A| > 1$ with its assigned bbm $m(A)$ denotes that all elements within A are incorporated in this bbm, but one cannot make any further assumptions on how $m(A)$ is distributed among them. There is no justification to split $m(A)$ into bbms of more elementary subsets of A. Shafer asserts that $m(\emptyset) = 0$, or equivalently that $bel(\Omega) = 1$ so that rules for belief combination and conditioning are normalised with appropriate scaling factors. Smets [19] discusses in more detail, that under the *open-world* assumption, $m(\emptyset) > 0$ can be justified. In the closed world, all hypotheses are specified and thus the conflict should be removed by normalisation. In an open-world, there might be additional unspecified hypotheses and thus the emptyset may have a mass greater than zero and normalisation can be omitted.

Belief functions are in one-to-one correspondence with **plausibility functions** $pl : 2^{\Omega} \rightarrow [0, 1]$. A plausibility function is defined by $pl(\emptyset) = 0$ and for all $A \subseteq \Omega, A \neq \emptyset$:

$$pl(A) = \sum_{B | A \cap B \neq \emptyset} m(B). \tag{2.4}$$

Plausibility and belief are related by:

$$pl(A) = bel(\Omega) - bel(\overline{A}). \tag{2.5}$$

where $\overline{A}$ is the complement of A relative to Ω (Smets [20]).

Compared to traditional probability theory, plausibility and belief form the upper and lower bound of a probability. A belief of an $A \in \Omega$ contains all belief masses that have to be assigned to A and the plausibility of A contains all belief masses that can *potentially* be assigned to A (Smets [20]).

$$bel(A) \leq P(A) \leq pl(A). \tag{2.6}$$

In the Dempster-Shafer model, fusion of two basic belief masses is realized through **Dempster's rule of combination**. It allows to combine two independent basic mass assignments m_1 and m_2 on the same frame of discernment. The **joint mass assignment** $m_{1,2}$ is calcu-

lated by:

$$m_{1,2}(\emptyset) = 0 \tag{2.7}$$

$$m_{1,2}(A) = (m_1 \oplus m_2)(A) = \frac{1}{1-K} \sum_{B \cap C = A \neq \emptyset} m_1(B) m_2(C) \tag{2.8}$$

$$K = \sum_{B \cap C = \emptyset} m_1(B) m_2(C) \tag{2.9}$$

K provides a measure for the conflict between the two mass sets since it corresponds to the sum of masses that have to be assigned to $\emptyset$. Thus, normalising with $\frac{1}{1-K}$ can be interpreted as ignoring the conflict. The fact that the conflict is ignored in Dempster's rule is problematic if there is a significant conflict in the mass assignments. This is because it leads to counterintuitive results. If two sources of evidence support an event E, one with a high mass of 0.99 and the other with a low mass of 0.01, the combination of them with ignoring the conflict results in $m(E) = 1$. $m(E) = 1$ is interpreted as full support for E, although the two sources reported opposite evidences. With Smets assumption that $m(\emptyset) \neq 0$, normalization is not required. The rule without normalisation is called **conjunctive rule of combination**.

Conditional probabilities as in Bayesian probability theory can also be expressed in the Dempster-Shafer model. They are important for expressing a (degree of) belief B in a context where A holds: $bel(B \mid A)$. Suppose bel quantifies the belief about the frame of discernment Ω and we learn that $\overline{A}$ is false. Then the corresponding mass assignment can be obtained by **Dempter's rule of conditioning**:

$$m(B|A) = \begin{cases} \frac{1}{1-K} \sum_{X \subseteq \overline{A}} m(B \cup X), & \text{if } B \subseteq A \subseteq \Omega, \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

$$K = bel(\overline{A})$$

Again, $\frac{1}{1-K}$ is a normalisation factor that can be omitted under the open-world assumption. According to the conditional mass assignment, these are the belief and plausibility formulas:

$$bel(B \mid A) = bel(B \cup \overline{A}) - bel(\overline{A}), \quad \forall B \subseteq \Omega \tag{2.11}$$

$$pl(B \mid A) = pl(A \cap B), \quad \forall B \subseteq \Omega \tag{2.12}$$

This way, a mass $m(B)$ given to B, is transferred by conditioning on A to $A \cap B$. Dempster's rule of conditioning (Equation 2.10) can be obtained without ever considering the concept of "combination of distinct pieces of evidence", hence it does not require any definition for distinctness, combination, or probability (Smets [20]).

Dubois and Prade [6] proved the following relations to combine a conditional belief or mass assignment given by m_1 and m_2:

$$m_{1,2}(A) = \sum_{B \subseteq \Omega} m_1(A|B) m_2(B) \tag{2.13}$$

$$bel_{1,2}(A) = \sum_{B \subseteq \Omega} bel_1(A|B) m_2(B) \tag{2.14}$$

$$pl_{1,2}(A) = \sum_{B \subseteq \Omega} pl_1(A|B) m_2(B) \tag{2.15}$$

For decision making processes, where one needs to choose (one) from multiple options, belief functions with mass assignments are not very useful. Most decision making processes are based on probabilities. The belief function model of Dempster-Shafer can be transferred into a probability model, where the Bayesian decision theory can be used. For every singleton element $w \in \Omega$ a probability can be calculated from belief functions with such a transformation.

Smets and Kennes [21] proposed the **pignistic transformation** to derive probability functions from belief functions. He defined the pignistic transformation by:

$$BetP(A) = \sum_{X \subseteq \Omega} \frac{|A \cap X|}{|X|} \frac{m(X)}{1 - m(\emptyset)} \tag{2.16}$$

Intuitively it means, that the masses of subsets of Ω are evenly distributed among its constituting singletons and can then be used as probabilities.

One other interesting property of belief functions in combination with this thesis is the ability to *discount* masses. Suppose someone tells you that the belief assignment for X is m, but you are not sure the source is reliable. If you believe at level α that the source is

reliable, and $1 - \alpha$ it is not, then your belief $\hat{m}$ in X becomes:

$$\hat{m}(A) = \alpha m(A), \forall A \neq X \tag{2.17}$$

$$\hat{m}(X) = 1 - \alpha + \alpha m(X) \tag{2.18}$$

This is called *discounting* by Shafer (Shafer [15], page 251 et seq.). Its name arises from the possibility to "discount" sources of information when one deems them not fully reliable. Beware that combination and discounting are not distributive, so the order with which they are applied is relevant.

In this thesis, the Dempster-Shafer theory of evidence is used in combination with predicate logic to evaluate formulas not only to *true* and *false*, but also to *unknown*. The result of an evaluated formula is annotated with masses, representing these three values. With such an annotation it is possible to build a ranking upon several formulas and determine the one which is most likely true.

2.3 Predicate Logic

In contrast to propositional logic, predicate logic offers the possibility to represent objects through *terms*. Predicate logic allows modelling structured objects and processes. Formally the alphabet of the predicate logic contains the following elements (see Chang and Lee [5]):

- *terms*
- *atoms*
- a finite or countably infinite set $\mathcal{F}$ of *function symbols* (i.e., the + in $+(1, 2)$)
- a finite or countably infinite set $\mathcal{R}$ of *relation (predicate) symbols* (i.e., $P, Q, \ldots$)
- a finite or countably infinite set $\mathcal{K}$ of *constant symbols*
- countably infinite set $\mathcal{V}$ of variables
- the set $\exists, \forall$ of *quantifiers*
- the symbols $\bot$ and $\top$
- the set of operators $\neg, \wedge, \vee$

Definition 1. *Terms are defined as follows; every variable and every constant is a term. If f is an n-place function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*
Definition 2. *If P is an n-place predicate symbol and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atom. The symbols $\bot$ and $\top$ are atoms. No other expressions are atoms.*
Definition 3. *The quantifiers $\exists$ and $\forall$ bind a variable within a formula. A variable is* bound *if and only if it occurs within the scope of a quantifier. An occurrence of a variable is* free *within a formula if and only if this occurrence is not bound.*
Definition 4. *A formula is defined by:*

- *Any* atom *is a formula.*
- *If F and G are formulas, then $\neg F$, $(F \wedge G)$, and $(F \vee G)$ are formulas.*
- *If F is a formula and x a variable, then $(\forall x)F$ and $(\exists x)F$ are formulas.*

Definition 5. *Clause*
A clause is a disjunction of literals, typically written as:

$$Head \leftarrow Body$$

where Head is a predicate and Body is a conjunction of literals.

For this thesis, predicate logic is important because conditions within ALICA are expressed in predicate logic. Conditions affect the actions of an agent. Thus, an agent wanting to track a team member should evaluate not only the conditions relevant for its own actions, but also those from its team members.

Resolution

A principle to prove that a first-order formula is unsatisfiable is *resolution* (Chang and Lee [5]). Resolution is an algorithm that can be easily implemented on a computer to automate the proof. Therefore, the formula is converted into a clause form. A formula in clause form is a conjunction of clauses. A conjunction of clauses is unsatisfiable if one of its clauses is unsatisfiable. Since a clause is a disjunction of literals, the clause is unsatisfiable if it is empty. *Resolution* tests if the empty clause can be created by applying operators that keep the semantic.
Definition 6. *Resolution Rule*
Given are two predicate logic clauses $D_1 = [p(s_1, \ldots, s_n), L_1, \ldots, L_m]$ and $D_2 = [\neg p(t_1, \ldots, t_n), K_1, \ldots, K_l]$ with $l, m, n \geq 0$. If the atoms $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ can

be unified by a most general unifier σ*,* $D = [L_1, \ldots, L_m, K_1, \ldots, K_n]\sigma$ *is called a* resolvent *of* D_1 *and* D_2 *upon* $p(s_1, \ldots, s_n)$ *and* $\neg p(t_1, \ldots, t_n)$*.* D *is created by applying the resolution rule on* D_1 *and* D_2*, whereas* $p(s_1, \ldots, s_n)$ *and* $\neg p(t_1, \ldots, t_n)$ *are the literals resolved upon. If* D_1 *is a clause, containing* $\bot$ *and* D *is created out of* D_1 *by removing all occurrences of* $\bot$ *from* D_1*, then* D *is called the* trivial resolvent *of* D_1*.*

Definition 7. *Factor rule*
Given a predicate logic clause:

$$D = [p(s_1, \ldots, s_n), p(t_1, \ldots, t_n), L_1, \ldots, L_m]$$

or

$$D = [\neg p(s_1, \ldots, s_n), \neg p(t_1, \ldots, t_n), L_1, \ldots, L_m]$$

with $m, n \geq 0$*, if* $p(_s, \ldots, s_n)$ *and* $p(t_1, \ldots, t_n)$ *can be unified by the most general unifier* σ*, then* $D' = [p(t_1, \ldots, t_n), L_1, \ldots, L_m]\sigma$ *and* $D' = [\neg p(t_1, \ldots, t_n), L_1, \ldots, L_m]\sigma$ *respectively are factors of* D*.* D' *is obtained by applying the factor rule on* D*.*

Definition 8. *Let* $F = \wedge_{i=1}^{n} D_i$ *a predicate logic formula in clause form, whereas* $D_i, 1 \leq i \leq n$ *are clauses.*

1. *The sequence*

$$\begin{array}{l} D_1 \\ \vdots \\ D_n \end{array}$$

   is a resolution derivation for F*.*

2. *If* S *is a resolution derivation for* F *and* D *is created by applying the resolution or factorisation rule on new variants of rows out of* S*, then* S *extended by* D *as the last row, is a resolution derivation for* F*.*

3. *A resolution derivation for* F *with the empty clause* $\bot$ *as the last row is called* resolution discount *for* F*.*

Definition 9. *Let* F *be a formula, then*

$$\vdash_\sigma F$$

is a resolution for F *with* σ *as a substitution.*

Definition 10. *Let F be a formula in predicate logic and G a clause form of $\neg F$. A resolution proof for F is a resolution discount for G. F is called a theorem of the resolution system, if no resolution proof for F exists.*

Resolution is important to this thesis, because it is the standard algorithm Prolog uses to evaluate programs. This algorithm is replaced in this thesis with one that allows for handling belief masses. In this thesis, a formula is not just true or false, but true, false and even unknown to a certain degree (expressed by a belief mass).

Formulas expressed as clauses in predicate logic, used in this section, have variables in their head which, after evaluation, specify their result. All other variables that only occur within the body of the clause need to be eliminated during the evaluation. Because the standard resolution does not provide belief masses for the variables, this thesis uses an approach that is based on Bayesian Networks, which are described in the next section.

2.4 Bayesian Networks

Bayesian Networks are used to reason with uncertainty in various systems. A Bayesian Network is a directed acyclic graph (DAG), whose nodes represent random variables. The edges of this DAG are conditional dependencies between those variables. Every variable depends on its parents, i.e., there is a directed edge from each parent to its children. Every node X_i contains a probability distribution $P(X_i \mid parents(X_i))$ (a conditional probability table, CPT), which quantifies the effect of the parent nodes towards X_i. All random variables together represent the uncertain knowledge. Instead of using a Bayesian Network, one could create the full joint probability distribution to represent the uncertain knowledge (Jensen [8]). Unfortunately in practise, this distribution is not very useful if the amount of variables strongly increases, because memory is limited and one often only knows the conditional probabilities so the full joint probability distribution cannot be created. An example for a Bayesian Network, borrowed from Russell and Norvig [14], is shown in Figure 2.1.

In the example depicted in Figure 2.1, an alarm system is shown. Whenever someone breaks into the house or there is an earthquake (the alarm system is based on a seismographic sensor), the alarm bell will ring. Once the alarm bell rings, the two neighbours John and Mary will call the house owner. Neither John nor Mary are 100% reliable, sometimes John mistakes his telephone ring with the alarm and thus calls the owner unnecessarily. Mary often hears loud music, so she does not hear the alarm bell.

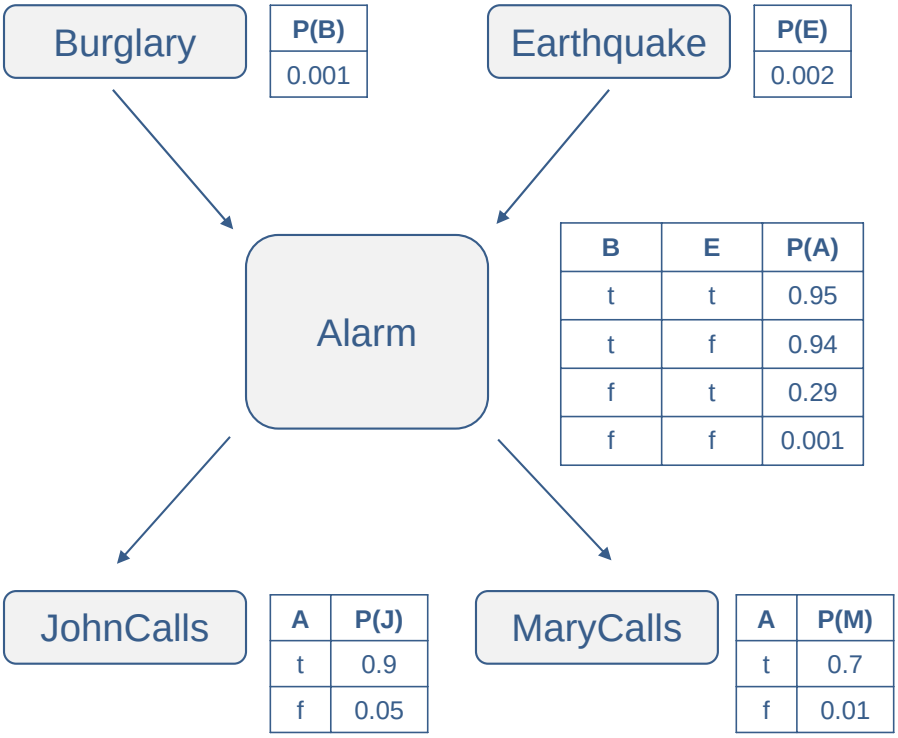


Figure 2.1: Bayesian Network Example

Every entry of the full joint probability distribution can be calculated from the Bayesian network. One such entry is the probability of a conjunction of concrete variable assignments, for all its variables, such as $P(X_1 = x_1 \wedge \ldots \wedge X_n = x_n)$, or shortly $P(x_1, \ldots, x_n)$. Thus an entry of the full joint probability distribution can be obtained from the network by Equation 2.19.

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i)) \tag{2.19}$$

$parents(X_i)$ represents the specific values of all parent variables. This way, every entry in the full joint probability distribution is described by the product of the corresponding elements in the CPTs of the Bayesian Network.

One example on how to use the Bayesian Network of Figure 2.1 will be given here. We want to calculate the probability that the alarm bell rang (a), but neither was there a burglary

(b), nor an earthquake (e), and John (j) as well as Mary (m) will call the house owner:

$$\begin{aligned} & P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) \\ & = P(j \mid a)P(m \mid a)P(a \mid \neg b \wedge \neg e)P(\neg b)P(\neg e) \\ & = 0,90 \times 0,70 \times 0,001 \times 0,999 \times 0,998 = 0,00062 \end{aligned}$$

Given a Bayesian Network it is possible to do inference on the data presented within the network. Usually an inference system should be able to calculate the conditional probability distribution for a set of query variables based on a certain observed event, i.e., an allocation of values to a set of evidence variables. If X is the query variable, E a set of evidence variables $E_1, \ldots, E_n$, e a certain observed event and Y stands for the hidden variables (non evidence variables) $Y_1, \ldots, Y_m$, then the complete set of variables is given by: $V = X \cup E \cup Y$. A request to the network could ask for the conditional probability distribution $P(X \mid e)$. Back to the example mentioned above, a request could be that there was a burglary and John, as well as Mary call the owner:

$$P(Burglary \mid JohnCalls = true, MaryCalls = true) = 0,284$$

For calculating the result of the example, one needs to *marginalise* over the hidden variables. Before explaining this, a short introduction to the *Bayes Theorem* (shown in Equation 2.20) is given:

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)} \tag{2.20}$$

The Bayes Theorem was derived by Bayes from the *product rule* which can be written as follows, since the conjunction is commutative:

$$\begin{aligned} P(A \wedge B) &= P(A \mid B)P(B) \\ P(B \wedge A) &= P(B \mid A)P(A) \end{aligned}$$

Thus, the probability of a variable B under the condition of variable A can be calculated with the knowledge of the a-priori probabilities of A and B and the probability of A under the condition of B, meaning the conditioning between variables can be *turned around*. Marginalisation sums up all probabilities for all possible values of a certain variable. A normalisation constant α is introduced for a distribution $P(A|B)$ that is independent from

the value of A and ensures that the sum is 1:

$$P(A_1 \mid B) = \frac{P(A_1 \wedge B)}{P(B)}$$
$$P(A_2 \mid B) = \frac{P(A_2 \wedge B)}{P(B)}$$
$$\text{hence } \alpha = \frac{1}{P(B)}$$

Considering α the conditional probabilities $P(A|B)$ and $P(B|A)$ can be summarized in Equation 2.21, in which $P(A, B)$ stands for the full joint distribution:

$$P(A \mid B) = \alpha_1 P(A, B);\ \alpha_2 P(A, B) = P(B \mid A) \tag{2.21}$$

With the normalisation factor α, the marginalisation over the hidden variables Y for a query variable X under the evidence e can be written as:

$$P(X \mid e) = \alpha P(X, e) = \alpha \sum_y P(X, e, y) \tag{2.22}$$

The example of the burglary with marginalisation and the normalisation factor α can be calculated by:

$$P(b \mid j, m) = \alpha P(B, j, m) = \alpha \sum_e \sum_a P(b, e, a, j, m) \tag{2.23}$$

Equation 2.23, together with Equation 2.19 results in:

$$P(b \mid j, m) = \alpha \sum_e \sum_a P(b)P(e)P(a \mid b, e)P(j \mid a)P(m \mid a) \tag{2.24}$$

The sums in Equation 2.24 can also be written in another way, such that the final result is:

$$P(b \mid j, m) = \alpha P(b) \sum_e P(e) \sum_a P(a \mid b, e)P(j \mid a)P(m \mid a) \tag{2.25}$$

Equation 2.25 shows how the probability of a burglary, under the assumption that John as well as Mary call the house owner, can be calculated with the help of the Bayesian Network shown in Figure 2.1.

As shown in this section, Bayesian Networks are a compact form to store the full joint probability distribution of a domain and they allow inferring unknown probabilities from knowledge within the network. In this chapter, the foundations of this thesis were presented to give the reader a basic overview about the techniques and approaches used in this thesis. The next section describes the problem addressed in this thesis. It explains the task definition and motivates the work of this thesis in the domain of RoboCup where it is evaluated in.

3 Problem Definition

This chapter describes the task of this thesis, together with a domain it is applied in, RoboCup. A brief description about the RoboCup domain and the importance of team play in it is given. Section 3.3 explains the CarpeNoctem Behaviour Engine, wherein the approach presented in this thesis can be used. It controls the behaviours of a team of robots for playing football.

3.1 Domain: RoboCup

RoboCup[1] is an international competition for teams of researchers around the world to improve robotics and artificial intelligence. It was founded by the RoboCup Federation with the main goal to provide an example domain and competition for researchers. Their secondary goal is having humanoid robots wining against the human world champion in football by the year 2050. Every year, there is a 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 frequently and the robots have to adjust themselves accordingly. It combines 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.

[1] http://www.robocup.org last accessed 03/10/2010
[2] http://www.robocup-german-open.de last accessed 03/10/2010
[3] http://www.fifa.com/worldfootball/lawsofthegame.html last accessed 03/10/2010

Because the approach in this thesis will be used in a Middle Size team, some rules for this league are explained here. A team consists of five 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 52cm x 52cm 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 robot's position based on the white field lines. Every year the rules are changed to become more similar 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 their common goal.

3.2 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. 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, for starting 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. Those 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 these situations cannot result in a direct goal, so the ball has to touch at least one other friendly robot besides the one scoring the goal.

In order to make team play possible the robots need to communicate with each other or use techniques such as plan recognition. Until now the robots of the team *CarpeNoctem* communicate some of their world model data and an internal representation of which actions they are executing. Based on the communicated information, each team member is aware of its team mates' actions and therefore they can coordinate their tasks.

However, communication is expensive in terms of the available bandwidth and unreliable in terms of delay. The environment might even prevent communication at all, e.g., because of other noisy radio signals. This needs to be kept in mind when designing a multi-agent system wherein the agents shall achieve common goals.

By looking on how humans coordinate their team play in football, it strikes out that they are told tactics or plans by their coach on how to behave in certain situations. Once the players are on the field they recognise these situations, i.e., during a Free-kick, they remember what their coach told them and transfer the tactic into their game play. The coach might have given them a plan such that some players stay in the defence, one executes the Free-kick and some others look for free areas to catch a pass from the Free-kick executor. They all know how to act according to the plan and to coordinate either by shouting short words or by watching the actions of their team member.

Hence, team play is a very important factor for a team of autonomous robots playing soccer. A lot of advantages arise for teams that have considered team play in their software, such as passes or a wall of defence for example. The CarpeNoctem RoboCup Team uses such a software implementation, which is described in the next section in more detail.

3.3 CarpeNoctem Behaviour Engine

The Behaviour Engine of the CarpeNoctem RoboCup team is an implementation of ALICA (Skubch et al. [18]), *A Language for Interactive Cooperative Agents*, that allows a user to specify how the agents should cooperate to achieve their common goal. The tracking approach presented in this thesis bases on ALICA and a proposed implementation targets on this Behaviour Engine. Thus, this section describes the Behaviour Engine, so the reader gains the background knowledge about ALICA and its implementation. The language consists of the following elements:

Definition. *Elements of ALICA (Skubch et al. [18], page 12)*

- *a logic $\mathcal{L}$, with language $\mathcal{L}(Pred, Func)$ meant to describe the agents' belief bases with a set of predicates $Pred$ and a set of function symbols $Func$.*
- *a set of Roles $\mathcal{R}$, which contains all available roles an agent can take on;*
- *a set of Plans $\mathcal{P}$, each describing a specific cooperative activity;*
- *a set of Tasks $\mathcal{T}$, each intuitively describing a function or duty within plans, meant to be fulfilled by one or more agents;*
- *a set of States $\mathcal{Z}$, a state occurs within a plan as a step during an activity;*
- *a set of Transitions $\mathcal{W} \subseteq \mathcal{Z} \times \mathcal{Z} \times \mathcal{L}$, connecting states within plans;*
- *a set of Synchronisations $\Lambda \subseteq 2^{\mathcal{W}}$, connecting transitions and denoting the need to synchronise certain actions;*
- *a set of Behaviours $\mathcal{B}$, encapsulating lower level actions, thus forming the atomic activities within ALICA;*
- *a top-level plan $p_0 \in \mathcal{P}$;*
- *a top-level state $z_0 \in \mathcal{Z}$;*
- *a top-level task $\tau_0 \in \mathcal{T}$;*
- *a set of functions,* PlanType*, each of type $2^{\mathcal{L}(Pred,Func)} \mapsto \mathcal{P}$, abstracting from a set of related plans.*

A directed graph is formed by the relationship between transitions and states for each plan. Edges are transitions and states represent its nodes. States contain plan types, thus this nesting forms a tree-like structure, called *plantree*. The top-level plan, p_0 is the root of this tree. During execution, an agent *moves* from state to state via transitions and thus the *plantree* can be seen as a hierarchical state machine.

Roles are assigned to agents according to their abilities, thus they reflect what an agent is capable to do. A *role* might be preferred for specific tasks or even not allowed to perform a concrete task by definition. Changing of *roles* only happens occasionally, once a team member leaves or joins the current active team, i.e., one agent needs to be replaced because of a failure.

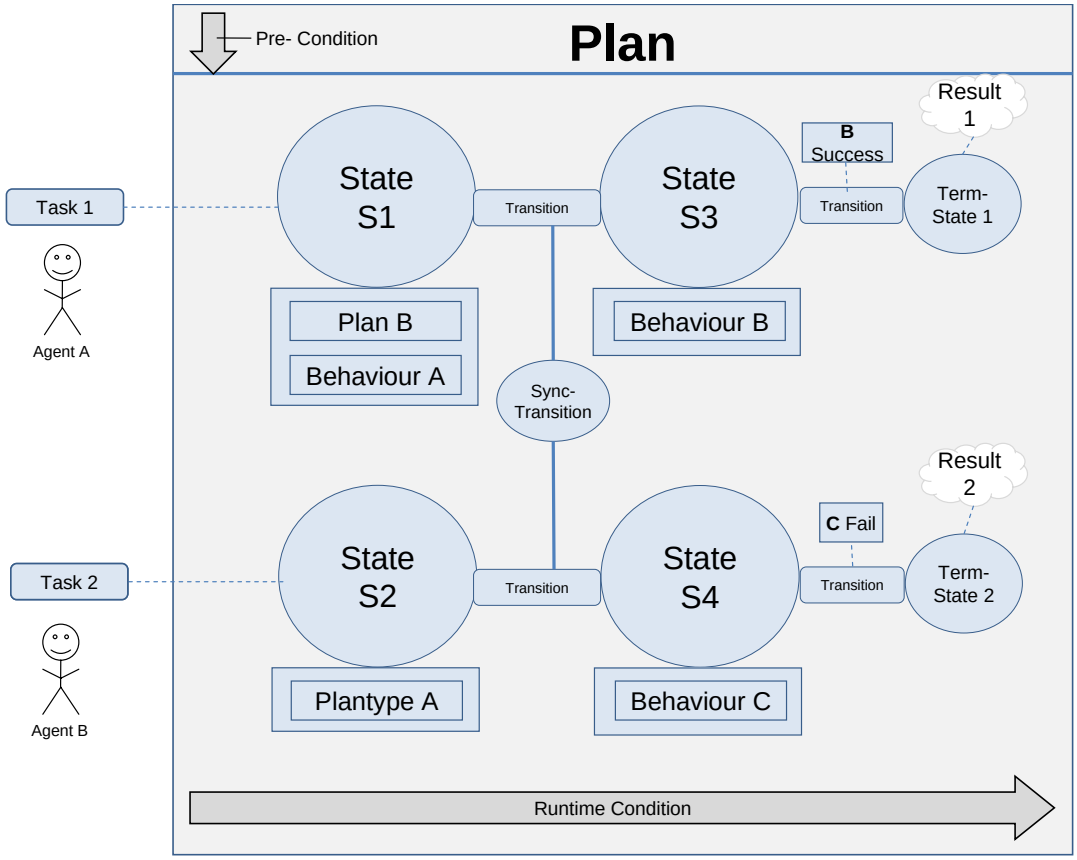


Figure 3.1: ALICA Plan Example

Figure 3.1 depicts a plan in ALICA and how the elements mentioned above are combined. Before this plan can be executed, each involved agent has to calculate an allocation of tasks to agents 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 tasks,
- Priorities of the roles of the agents for the two tasks,
- Pre- and Runtime-condition of the plan.

These items are encapsulated within a *utility function* that is attached to each plan. An agent may change its task on a plan more frequently than its role. It makes this decision by evaluating the plan's *utility function* on a regular basis and possibly switch its task, if

the utility would be higher. The result of an evaluated *utility function* is a number in the interval $[-1..1]$ and stored on the allocation.

An allocation contains information about which agent is committed to which task for a given plan. $\text{In}(a, p, \tau, z)$ denotes that agent a is committed to task τ in plan p and that a is currently in state z. A conjunction of $\text{In}(a, p, \tau, z)$ with the same plan is called an allocation. The allocation is considered to be valid iff the cardinalities on the tasks are not violated, the pre- and runtime-conditions hold assuming the allocation, the utility is ≥ 0 and it is consistent with the axioms of ALICA (Skubch et al. [18], page 16 et seq.). If the allocation is valid and believed to be the same for all participating agents, they agree on who is doing which part of the plan.

A state contains one or more plan types. An agent being in this state during execution has to select a plan from each plan type and execute it. In the plan of Figure 3.1, State $S2$ contains PlanType A and the agent in this state has to select and execute exactly one plan out of it. Because each plan has a utility function attached to it, an agent is able to calculate the maximum possible utility for each of those plans and select the one with the highest utility. Since *utility functions* are mostly based on uncertain or even outdated environment data their result might differ for distinct agents and a failure in their team play might occur if they execute different plans from the given PlanType or compute different task allocations for a plan.

For stabilizing agreements on all allocations in the agents *plantrees*, each agent communicates its currently active *plantree* to its team members. Received *plantrees* from team members are then taken into account while updating allocations. This updating process is based on regular communication and supports keeping the agents to believe the same task allocations.

Besides this regular communication, there are *Synchronisations* which may exist between two or more transitions, requiring the agents to synchronise before they are allowed to progress along these transitions. This synchronisation is achieved through explicit communication specified by a synchronisation protocol. Agents in states, whose outgoing transition is connected to a synchronisation point, participate in a synchronisation. All those agents have to achieve a mutual belief about that their team members are in their corresponding states and believe the condition for their transition holds. Formula 3.1 shows this, where ϕ_i

denotes the condition of one agent.

$$\text{MBel}(\text{In}(a_i, p, \tau_i, z_i) \wedge \phi) \tag{3.1}$$
$$\phi = \phi_1 \wedge \ldots \wedge \phi_n$$

The implementation of ALICA is written in the programming language C#, using its open source implementation Mono[4]. It is separated into sub modules as shown in Figure 3.2. The

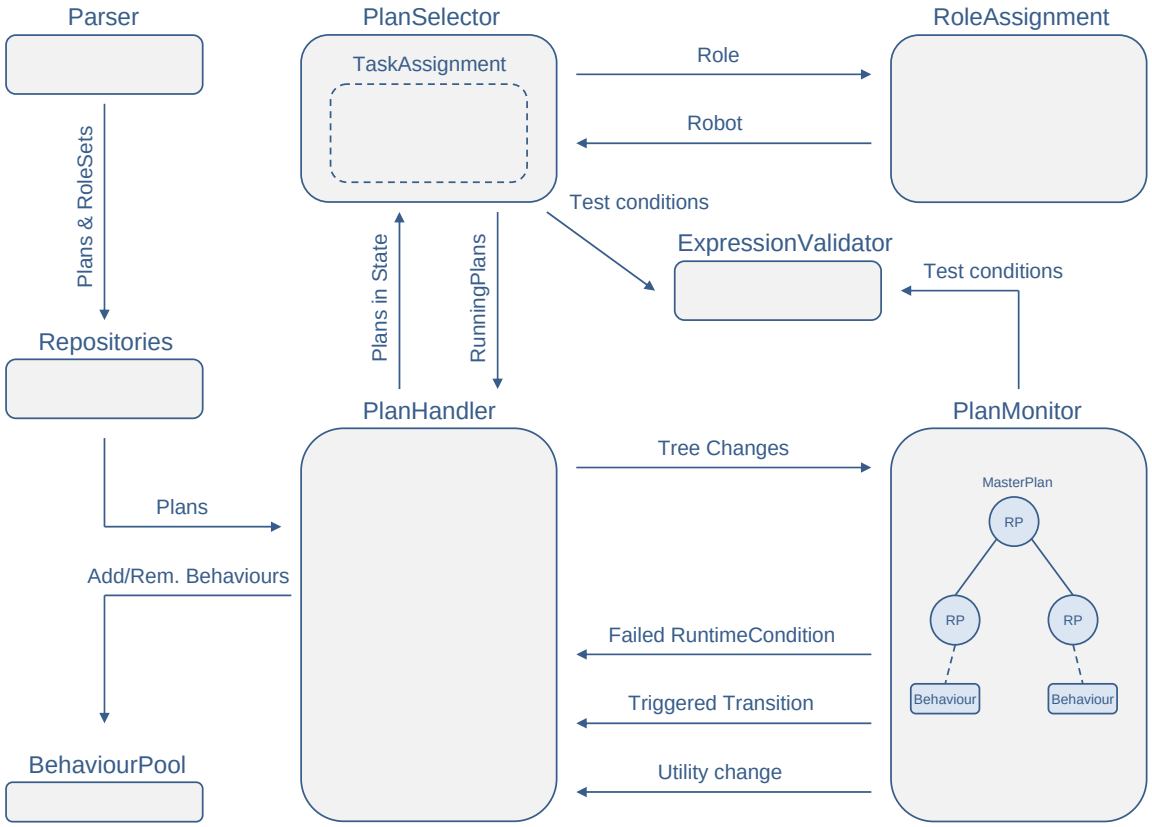


Figure 3.2: CarpeNoctem Behaviour Engine

two most important modules are the *PlanMonitor* and *PlanHandler*. The former monitors the conditions on plans and transitions and the later applies the changes of plans and states. The *PlanSelector* allocates team members to tasks on plans and selects plans out of plan types. Plans are stored within *Repositories* and behaviours are started and stopped by the *BehaviourPool*. The *ExpressionValidator* executes pre-compiled C# code that returns

[4]http://www.mono-project.com last accessed 03/29/2010

either true or false for a certain condition. The *RoleAssignment* assigns roles to robots according to their abilities and capabilities.

After the short introduction about RoboCup and the CarpeNoctem Behaviour Engine, the next section motivates and explains the task of this thesis.

3.4 Task Definition and Motivation

The goal of this thesis is to further improve the coordination and team play of agents, executing an implementation of ALICA. Within the CarpeNoctem Behaviour Engine described in the previous section, agents exchange information about their current state in plans and tasks they are executing. This is necessary for their coordination, since they need to agree on which agent of the team performs which task. However communication is expensive and unreliable, because it takes time until a message is created, encoded, send, received, and decoded, and the communication may also be interrupted. If one wants to be sure that a message was received at the other end, a protocol is necessary that exchanges more than just the one message containing the information.

This thesis aims on coordinating a team of agents with less communication effort. Therefore, each member of the team should track its team members in the plans they execute. In ALICA, all agents are aware of a common plan library and thus all members of a team know all plans. Once an agent has received information about the states a team member is in, it can use the plan library to try to predict what it might do next. This can be done by evaluating all conditions of the plans and states it is in, with respect to its view of the environment. In order to do this, a view of the environment for every team member (team world model) is needed. Many hypotheses can be created about what this team member is executes, because there usually are plenty of possible state changes and/or run-time conditions, which all lead to different alternatives.

The hypotheses need to be combined into team hypotheses. A team hypothesis describes what a team of agents is currently doing. A way needs to be found that makes it possible to rank such team hypotheses, because an agent has to make a decision on which hypothesis is the best and which others are still worth to track. Once an agent has team hypotheses, they need to be updated. The update can either result from received information about the current states of the team members, or from evaluating relevant conditions.

Conditions in ALICA plans are central for tracking agents in plans, since they are attached

to transitions and define when an agent *moves* from one state to another. They also define a pre- or runtime-condition that can be evaluated to test if a team hypotheses is valid. An evaluation to just *true* or *false* is not helpful for a tracker, because it needs a measurement unit to determine if one possibility is better or worse than another. Thus the evaluation of such conditions needs to be extended to produce such a measurement unit.

Team hypotheses, that are created and used by the tracking approach, should be easily extendable to include information obtained from the observed environment, also known as *Plan recognition*. This is because plan recognition can further improve the tracking.

This chapter explained the task of this thesis and its foundations. In the next chapter some related work is presented about tracking agents in multi-agent systems, and how others proposed probabilistic logic programs.

4 Related Work

Plan recognition and tracking of agents within plans are related topics. Both methods can improve the coordination of agents within a multi-agent system. Plan recognition infers which plans an agent executes by observing its actions and taking the environment into account. Tracking predicts which actions an agent executes within a plan. Unfortunately, they both suffer from uncertainty, which needs to be taken into account when designing a system that uses plan recognition or tracking. Some existing approaches which are similar to the approach in this thesis are presented in this chapter.

Avrahami-Zilberbrand and Kaminka [1] propose SBR (Symbolic Plan recognition) for recognizing a single agent's plans. It bases on a single-root-directed graph, where nodes are plan-steps and edges are either top-down edges that decompose plan-steps into sub-steps or sequential edges that specify the execution order in time. Every plan has conditions of observable features attached that need to hold in order to assume the agent is executing that plan. At any given time the agent executes a path from root to leaf of the plan graph. Such a path is considered as the internal state of the agent, which maybe changed if the agent follows a sequential edge to the next plan-step or interrupts the plan and selects a new (first) plan.

An extension of SBR for multi-agent systems is also proposed by Avrahami-Zilberbrand and Kaminka [2]. It tracks groups of agents which may split into smaller groups. As a plan recognizer, SBR (Avrahami-Zilberbrand and Kaminka [1]) is used for each agent to filter hypotheses. A Dynamic Hierarchy Group Model (DHGM) is created out of all *current state hypotheses* from all agents. Agents executing the same plan-steps, identified by the plan recogniser, are treated as one group and represent a node within the DHGM. Nodes have branches to sub-groups, which represent that a group has been split and their members execute different plan-steps. Branches are merged if an agent returns to its group or joins other sub-groups. All nodes point to plans within the plan library. To update the DHGM,

an algorithm processes it from bottom up. The SBR is executed for each leaf, which possibly creates new branches. Afterwards a merging algorithm merges branches so agents with the same path are in the same group.

In this thesis we also create single hypotheses for each agent within a team and merge them to a team hypothesis. Since our plans span a hierarchy too, our team hypotheses also contain such a hierarchy of plans. In contrast to the multi-agent SBR, we are able to rank hypotheses and can keep only the ones that suit best. This has the advantage that we have probabilities for the agents executing certain plan-steps and thus we can easily switch to the second best hypothesis if the first one turns out to be wrong.

Tambe [27] proposes $RESC_{team}$ that represents only one coherent team hypothesis. It is based on RESC (REal-time Situated Commitments) (Tambe and Rosenbloom [28]). In RESC the tracker executes a model of the agent being tracked (trackee), where it matches the trackee's actions against the model. The model is executed with the trackee's perspective of the environment. In order to do that in real-time, RESC commits to one single execution path, this is determined using a heuristic. If this leads to a tracking error, a repair rule is invoked, which resolves commitments from bottom up through the model (via backtracking) until it matches again and commits to an alternative. As an extension, $RESC_{team}$ uses team states and team operators. Team states track a team's joint state, which contains a *shared* part that is the same for all team members and a *divergent* part where the models of the agents differ. A state of an agent contains its view of the environment. Team operators represent the team's joint commitment to a joint activity; such an activity is specified by roles. These team operators span a hierarchy that specifies each agent's actions and is also capable of representing sub-teams. A tracking agent commits to a hierarchy of team operators and applies team states to it to predict the team's actions, i.e., it tries to match activities to the agent's environments. Sub-teams are treated as a whole and thus a change in a sub-team member's activity is assumed to match the whole sub-team.

$RESC_{team}$ has a lot of things in common with the tracking approach in this thesis. We also create team hypotheses that reflect the current actions of a team. Inside those team hypotheses we keep track of each agent's activities and apply their view of the environment to the conditions of the plans they are executing. However we do not stick to just one hypothesis, but keep track of several alternatives to avoid backtracking in case of tracking failures. $RESC_{team}$ also does not specify how the heuristic to determine a single execution path of an agent works. We propose an approach that uses belief theory to decide of one hypothesis is better than another.

A very similar approach to [27], by Kaminka and Tambe [9], is a reactive plan-recognition algorithm: RESL (REal-time Situated Least-commitments). It creates hypotheses where each team member is within a plan and combines them into a single team hypothesis. To do this, it expands the whole plan-library for each team member and tags all paths matching the observed behaviour of that team member. Paths (hypotheses) that are deemed inappropriate, are eliminated by heuristics, which is similar to the team tracking in Tambe [27].

One step further is the proposition of Tambe [26] for recursive agent tracking. It offers the possibilities to not only track other agents, but also what those other agents might think of the tracking agent. This way, in scenarios like military combats a pilot could track its enemy and base its decisions on what the enemy thinks about him. At first, a model of an agent A is created, then the model of what agent A thinks about another agent B (AB) can be build. This can even be extended further to a model of what A thinks that B thinks about A (ABA). To keep the effort low, the approach uses model sharing with pointers, so models that are equal are shared and additional models are only created if they diverge.

Other plan recognition systems, such as Lesh and Etzioni [11] expand a graph with actions and goals from a plan-library and apply rules for each observed action to remove goals until only one remains. According to their paper it is fast, provably sound and runs in polynomial-time. However, this approach is just for recognising plans of one single agent, whereas ours targets on multi-agent systems. Recognizing teams of agents and inferring their actions from geometric positions is addressed in STABR (Simultaneous Team Assignment and Behavior Recognition) (Sukthankar and Sycara [24]). It also maps agents into teams which may change over time. Sukthankar and Sycara [25] also propose a mechanism to rank and prune hypotheses by incorporating local temporal dependencies between the team member's actions. A different approach by Intille and Bobick [7] recognises team-tactics for football players by focusing on the interactions between agents instead of treating each agent as an individual. They use belief networks for representing and recognising individual agent goals from visual evidence. Such belief networks are similar to what we use to eliminate variables while evaluating conditions with variables annotated with confidences that are used as belief masses.

Ng and Subrahmanian [13] introduce a logic programming language where truth values are interpreted probabilistically. It extends the annotated logics, introduced by Blair and Subrahmanian ([4, 22, 23]) to allow conjunctions and disjunctions to be annotated and allow annotations to be closed intervals of truth values instead of single values. Those intervals

reach from zero to one ($[0, 1]$), whereas the interval $[0, 0]$ denotes that the annotated formula has a probability of zero and thus it is false. Ng and Subrahmanian state definitions on how the intervals of truth values can be combined. The approach of annotating formulas with intervals of truth values is similar to the evaluation of conditions in this thesis. Instead of annotating formulas with intervals of truth values, we annotate certain assignments of variables with confidences. These confidences are used as masses for the Dempster-Shafer theory of evidence and thus we can tell the amount of truth that results from a formula, given a specific combination of variable assignments. Further we can also express how likely a formula is false and even not knowing the result of a formula. The ability to express that information is unknown is necessary in a tracking approach for multi-agent systems. This is because one cannot always decide whether a condition is true or false, if certain information is unknown.

Tracking and plan recognition approaches have to deal with the problem that they need to find the correct hypothesis for the team out of many alternatives. Avrahami-Zilberbrand and Kaminka [3] propose an approach that uses utility functions to rank those hypotheses so a plan recogniser or tracker can work with the best one. Unfortunately their utility functions are based on probabilities that are annotated on the transitions which change the behaviour of an agent, by humans, prior to execution. This thesis presents an approach that does not need such a human interaction, because probabilities are calculated based on environment observations with confidences attached. The next chapter describes this approach.

5 Approach

This chapter focuses on a tracking approach that is able to weight between several possible hypotheses, but without the need to explicitly tag alternatives with probabilities. It is divided into two main sections. Section 5.1 describes an implementation to evaluate conditions in ALICA (see Section 3.3 for how those conditions are used). This implementation is mainly based on the Dempster-Shafer theory of evidence and is written in the logic programming language *Prolog*, using the *ECLiPSe* [1] implementation of it. We use Prolog because we need to evaluate first-order formulas, which can easier be implemented in a logic programming language than C#, in which the rest of the CarpeNoctem Behaviour Engine is written.

Section 5.2 describes a possible tracking implementation for the Carpenoctem Behaviour Engine that is based on the evaluator presented in Section 5.1. It describes how hypotheses about a team's actions can be created and updated. While creating this tracking approach, the *Synchronisation Points* of ALICA were implemented in the CarpeNoctem Behaviour Engine. They give precise evidence for the state of a plan, an agent participating such a synchronisation, inhabits. At the end of Section 5.2 a brief overview is given on how one can possibly include plan recognition into this tracking approach.

5.1 Evaluation of Conditions

In this section, the implementation of an evaluator for first-order formulas using the Dempster-Shafer theory of evidence is explained. The main idea behind this evaluator is evaluating predicates not only to either *true* or *false*, but also to *unknown*. As in the standard resolution, clauses are evaluated and variables may be bound. One problem for a

[1] http://www.eclipse-clp.org last accessed: 03/06/2010

successful predicate evaluation is that not all variables are known during their evaluation. In addition, we can not only tell if a formula is *true*, *false*, or *unknown*, but also assign masses to these values. We use normalised masses and thus the mass of 1 is split upon the three values. In the RoboCup domain, where the evaluator presented in this section will be used, sensor noise leads to imprecise data, which we can express with a mass for *unknown*. For traditional predicate logic that relies on complete information, such a scenario would lead into problems. Imagine that the position of the ball is unknown and a predicate should calculate the distance from a known robot position to it. Such a predicate might look like: $dist(Pos, Ball, D)$. Since $Ball$ and D are free variables the evaluation process will try to bind them, figure out it cannot do that and either fail or evaluate the predicate to false, depending on the logic programming language used. To overcome this problem, the evaluator presented here will evaluate such a predicate to *unknown* and treat the variables $Ball$ and D as if they have an *unknown* value, too.

5.1.1 Expressing Lack of Knowledge

The background for expressing lack of knowledge is unreliable information from a robot's sensors about the environment. Also, the tracking approach in Section 5.2 needs the ability to express that a robot does not know what its team member is currently doing. Thus the result of an evaluated formula will be a *belief vector*, shown in Equation 5.1, attached with the bound variables.

$$belVec = (m_{true}, m_{false}, m_{unknown})^T \ , \ \sum m \in belVec = 1 \tag{5.1}$$

m_{true}, m_{false}, and $m_{unknown}$ are called *true-part*, *false-part*, and *unknown-part* in this thesis, respectively. These belief vectors can be combined by the $\wedge$- and $\vee$-operators (SIMON and WEBER [17]), using the matrices in Definition 5.2 or 5.3, respectively.

$$mt_{true} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, mt_{false} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}, mt_{unknown} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{5.2}$$

The conjunction $\vec{v_{12}} = v_1 \wedge v_2$ between two belief vectors $\vec{v_1} = \left(m_{1t}, m_{1f}, m_{1u}\right)$ and $\vec{v_2} =$

$\left(m_{2t}, m_{2f}, m_{2u}\right)$ can be calculated as follows:

$$\begin{aligned} \vec{v_{12t}} &= \vec{v_1}^T * mt_{true} * \vec{v_2} \\ \vec{v_{12f}} &= \vec{v_1}^T * mt_{false} * \vec{v_2} \\ \vec{v_{12u}} &= \vec{v_1}^T * mt_{unknown} * \vec{v_2} \end{aligned}$$

Similar to the conjunction, one can calculate the disjunction $\vec{v_{12}} = v_1 \vee v_2$ between two belief vectors $\vec{v_1} = \left(m_{1t}, m_{1f}, m_{1u}\right)$ and $\vec{v_2} = \left(m_{2t}, m_{2f}, m_{2u}\right)$ as follows:

$$mt_{true} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, mt_{false} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, mt_{unknown} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{5.3}$$

$$\begin{aligned} \vec{v_{12t}} &= \vec{v_1}^T * mt_{true} * \vec{v_2} \\ \vec{v_{12f}} &= \vec{v_1}^T * mt_{false} * \vec{v_2} \\ \vec{v_{12u}} &= \vec{v_1}^T * mt_{unknown} * \vec{v_2} \end{aligned}$$

Negation of a belief vector is defined below:

$$\begin{aligned} \vec{bv} &= (m_{true}, m_{false}, m_{unknown})^T \\ \neg\vec{bv} &= (m_{false}, m_{true}, m_{unknown})^T \end{aligned} \tag{5.4}$$

Section 5.1.2 explains how an evaluator for formulas was implemented using these belief vectors.

5.1.2 Implementation

This section presents the implementation of a formula evaluator in the logic programming language Prolog. Section 5.1.2.1 explains the data structures used in the implementation, Section 5.1.2.2 describes the process of the predicate evaluation and Section 5.1.2.3 presents an algorithm that is able to remove variables that only appear within a clause' body but not in its head.

5.1.2.1 Data Structures

The core of the evaluator implementation are two data structures, an *assignment* list and a *dependency* list, both encapsulated into a solution. Thus, a solution contains a list of assignments and a list of dependencies. An assignment contains a belief vector as described in Section 5.1.1 and a list of variables with their values. Equation 5.5 shows an assignment.

$$Assign = \left((m_{true}, m_{false}, m_{unknown})^T, [X_1 = v_1, \ldots, X_n = v_n] \right) \tag{5.5}$$

A dependency contains two entries and a *mass*. The first entry represents a variable and its value. The second is a list that contains variables and their values, the variable in the first entry depends on. m specifies a mass value which belongs to the variable allocation given in this dependency. Equation 5.6 shows a dependency.

$$Dep = (X = v, [Y_1 = w_1, \ldots, Y_n = w_n], m) \tag{5.6}$$

Definition 11. *Predicates are divided into three groups:*

1. *World model predicates $W(\vec{X}) \in \mathcal{W}$, that obtain values from the robot's world model data which are usually annotated with a confidence. These predicates bind variables and also provide a confidence for them. They may return different results for each call and cannot fail. They are either true or unknown.*
2. *Traditional predicates $T(\vec{X}) \in \mathcal{T}$, that are atomic in the sense that they are not evaluated by our approach, but by the normal Prolog interpreter.*
3. *Derived predicates $D(\vec{X}) \in \mathcal{D}$ that are defined by clauses visible to our evaluator.*

All predicates may only contain one single free variable in their parameters, i.e., all other parameters have to be bound prior to their evaluation.

This implementation limits the free variables within predicate parameters to be just one single variable. This is because such dependencies with more than one variable in the first entry are difficult to handle when removing variables, which will be explained in Section 5.1.2.3.

Definition 12. *Evaluator*
Our evaluator is specified by:

$$E\left(\mathcal{Q}, \mathcal{P}, \mathcal{W}, \mathcal{BK}\right) = \vec{bv}$$

iff

$$eval(\mathcal{Q}, [], []) = (\mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}),\ \textit{with}\ [], []\ \textit{being empty lists of assignments and dependencies}$$

All variables in the assignments of $\mathcal{L}_\mathcal{A}$ are removed (see Section 5.1.2.3), thus $\mathcal{L}_\mathcal{A}$ contains only one assignment and $\mathcal{L}_\mathcal{D}$ is an empty list:

$$\mathcal{L}_\mathcal{A} = [As_1]\ ;\ \mathcal{L}_\mathcal{D} = []$$

and

$$As_1 = \left(\vec{bv}, []\right)$$

$\mathcal{Q}$ defines a query to the evaluator, a conjunction of literals. $\mathcal{P}$ is a program that contains a set of clauses, whose heads are in $\mathcal{D}$. The robot's world model is specified by $\mathcal{W}$. The background knowledge is specified by $\mathcal{BK}$ and it contains callable predicates of type $\mathcal{T}$. $\mathcal{W}$ and $\mathcal{BK}$ are omitted as parameters for $eval(\mathcal{Q}, \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D})$, because they are defined to always be available to the evaluator. $\mathcal{L}_\mathcal{A} = [As_1, \ldots, As_n]$ is a list of assignments and $\mathcal{L}_\mathcal{D} = [Dep_1, \ldots, Dep_n]$ is a list of dependencies.

5.1.2.2 Formula Evaluation

Predicate evaluation works as follows: First the bound and free variables of a predicate's parameters are identified. Free variables that will be calculated and bound by the predicate depend on the ones that are already bound. A dependency is created for every solution. In addition to this dependency an assignment is created that contains the newly bound variable with its value and a belief vector expressing that this assignment is true or false. If the free variable cannot be bound by the predicate and thus it stays in the unknown state, the belief vector of the assignment is set to unknown. When there are no solutions for the predicate, given the bound variables, the belief vector of the assignment is set to false. A formal description for all three predicate types is given in Definition 16.

Before giving the definition for the evaluation of different predicates types, we need to define the *application* of an assignment to a predicate, *extending* an assignment, and *combining* two assignments disjunctively. This is defined in Definitions 13, 14, and 15, respectively.

Definition 13. *Applying an assignment to a predicate:*
Let $A = \left(\vec{bv}, [X_1 = v_1, \ldots, X_n = v_n]\right)$ be an assignment and $P(Y, X_1, \ldots, X_n)$ a predicate,

then A can be applied to $P(Y, X_1, \ldots, X_n)$ with

$$P(Y, X_1, \ldots, X_n)\, A = P(Y, v_1, \ldots, v_n)$$

A acts as a substitution.

Definition 14. *Extending assignments:*
Given two assignments A and A', A can be extended by A' to E with $A \odot A' = E$, if they do not contain contrary values for the same variable.

$$A = \left(\vec{bv}, [X_1 = v_1, \ldots, X_n = v_n]\right)$$

$$A' = \left(\vec{bv'}, [X'_1 = v'_1, \ldots, X'_n = v'_n]\right)$$

$$E = \left(\vec{bv} \wedge \vec{bv'}, [X_1 = v_1, \ldots, X_n = v_n, X'_1 = v'_1, \ldots, X'_n = v'_n]\right)$$

The variable-value lists of A and A' are concatenated into one single list and the belief vectors are combined by the $\wedge$-operator. Double variables are omitted.

Definition 15. *Combining assignments disjunctively:*
Given two assignments A and A', then $A \vee A' = E$:

$$A = \left(\vec{bv}, [X_1 = v_1, \ldots, X_n = v_n]\right)$$

$$A' = \left(\vec{bv'}, [X_1 = v_1, \ldots, X_n = v_n]\right)$$

$$E = \left(\vec{bv} \vee \vec{bv'}, [X_1 = v_1, \ldots, X_n = v_n]\right)$$

Definition 16. *Evaluation of a single predicate*
Assume eval $\left(P(\vec{X}), \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right)$ is called.

1. *Let $P(\vec{X}) = W(\vec{X})$ be a predicate in $\mathcal{W}$, X a free variable in $\vec{X}$ and $Y_1, \ldots, Y_n$ with values $w_1, \ldots, w_n$ variables in $\vec{X}$, bound by $W(\vec{X})As_j$ with an assignment As_j out of $\mathcal{L}_\mathcal{A}$. X will be bound to world model data v by $\vdash_{\sigma_j} W(\vec{X})A_j$. Then eval $\left(W(\vec{X}), \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right)$ results in one assignment $Assign_j$ with a substitution $\sigma_j = [X = v_j]$ for every solution v_j of X:*

$$Assign_j = \begin{cases} \left((1,0,0)^T, [X = v_j, Y_1 = w_1, \ldots, Y_n = w_n]\right) & \text{if } \vdash_{\sigma_j} W(\vec{X})A_j \\ \left((0,0,1)^T, [X = X, Y_1 = w_1, \ldots, Y_n = w_n]\right), & \text{otherwise} \end{cases}$$

and one dependency Dep for every solution v_j of X,

$$Dep_j = \begin{cases} (X = v_j, [Y_1 = w_1, \ldots, Y_n = w_n], m) & \text{if } \vdash_{\sigma_j} W(\vec{X})A_j \wedge (X = v_j) \in \sigma_j \\ (X = X, [Y_1 = w_1, \ldots, Y_n = w_n], m) & \text{otherwise} \end{cases}$$

m is the confidence for variable X obtained from the world model data. Thus

$$\begin{aligned} eval\left(W(\vec{X}), \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) &= \left(\mathcal{L}'_\mathcal{A}, \mathcal{L}'_\mathcal{D}\right) \\ \mathcal{L}'_\mathcal{A} &= [As_1 \odot Assign_j, \ldots, As_n \odot Assign_j] \\ \mathcal{L}'_\mathcal{D} &= \mathcal{L}_\mathcal{D} \cup [Dep_1, \ldots, Dep_n] \end{aligned}$$

2. *Let $P(\vec{X}) = T(\vec{X})$ be a predicate in $\mathcal{T}$, X a free variable in $\vec{X}$ and $Y_1, \ldots, Y_n$ with values $w_1, \ldots, w_n$ variables in $\vec{X}$, bound by $T(\vec{X})As_j$ with an assignment As_j out of $\mathcal{L}_\mathcal{A}$. $eval\left(T(\vec{X}), \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right)$ results in an assignment $Assign_j$ with a substitution $\sigma_j = [X = v_j]$ for every solution v_j of X:*

$$Assign_j = \begin{cases} \left((1,0,0)^T, [X = v_j, Y_1 = w_1, \ldots, Y_n = w_n]\right) & \text{if } \vdash_{\sigma_j} T(\vec{X})A_j \\ \left((0,1,0)^T, [X = X, Y_1 = w_1, \ldots, Y_n = w_n]\right) & \text{if } \neg\,(\exists X)\, T(\vec{X})A_j \\ \left((0,0,1)^T, [X = X, Y_1 = w_1, \ldots, Y_n = w_n]\right), & \text{otherwise.} \end{cases}$$

and a dependency

$$Dep_j = \begin{cases} (X = v_j, [Y_1 = w_1, \ldots, Y_n = w_n], 1) & \text{if } \vdash_{\sigma_j} T(\vec{X})A_j \wedge (X = v_j) \in \sigma_j \\ (X = X, [Y_1 = w_1, \ldots, Y_n = w_n], 1) & \text{otherwise.} \end{cases}$$

The mass is set to 1, because predicates of this type are seen to only produce results with no uncertainty. If there is no free variable in $\vec{X}$ the creation of a dependency is omitted. Thus

$$\begin{aligned} eval\left(T(\vec{X}), \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) &= \left(\mathcal{L}'_\mathcal{A}, \mathcal{L}'_\mathcal{D}\right) \\ \mathcal{L}'_\mathcal{A} &= [As_1 \odot Assign_j, \ldots, As_n \odot Assign_j] \\ \mathcal{L}'_\mathcal{D} &= \mathcal{L}_\mathcal{D} \cup [Dep_1, \ldots, Dep_n] \end{aligned}$$

3. *Let $D(\vec{X})$ be a predicate in $\mathcal{D}$, such that $S = \{D(\vec{X_1}) \leftarrow B_1, \ldots, D(\vec{X_n}) \leftarrow B_n\}$ is the*

set of clauses with $D(\vec{X})$ as head. Then, if

$$eval\left(B_j, \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) = \left(\mathcal{L}'_{\mathcal{A}j}, \mathcal{L}'_{\mathcal{D}j}\right)$$

then variables in $\mathcal{L}'_{\mathcal{A}j}$ and $\mathcal{L}'_{\mathcal{D}j}$ that do not appear in $\vec{X}$ are removed as described in Section 5.1.2.3, which results in $\left(\mathcal{L}''_{\mathcal{A}j}, \mathcal{L}''_{\mathcal{D}j}\right)$ and

$$\begin{aligned} eval\left(D(\vec{X}), \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) &= \left(\mathcal{L}'''_\mathcal{A}, \mathcal{L}'''_\mathcal{D}\right) \\ \mathcal{L}'''_\mathcal{A} &= \left(\mathcal{L}''_{\mathcal{A}1} \vee \ldots \vee \mathcal{L}''_{\mathcal{A}n}\right) \\ \mathcal{L}'''_\mathcal{D} &= \mathcal{L}_\mathcal{D} \cup \left[\mathcal{L}''_{\mathcal{D}1} \vee \ldots \vee \mathcal{L}''_{\mathcal{A}n}\right] \end{aligned}$$

A conjunction of two formulas ϕ_1 and ϕ_2 is evaluated as described above, but ϕ_2 may obtain input from the result of ϕ_1. Every assignment created by ϕ_1 is applied to ϕ_2 before ϕ_2 is evaluated. All resulting assignments out of the evaluation for ϕ_2 need to be combined with the input assignment, which means the input assignment is extended by the newly calculated variable and the new belief vectors are the result of the conjunctive combination of the two belief vectors from the involved assignments. Definition 17 describes this.

Definition 17. *Evaluation of a conjunction*
A conjunction of two formulas $\phi \wedge \phi'$ is evaluated as follows:

$$\begin{aligned} eval\left(\phi \wedge \phi', \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) &= \left(\mathcal{L}''_\mathcal{A}, \mathcal{L}''_\mathcal{D}\right) \\ eval\left(\phi, \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) &= \left(\mathcal{L}'_\mathcal{A}, \mathcal{L}'_\mathcal{D}\right) \\ eval\left(\phi', \mathcal{L}'_\mathcal{A}, \mathcal{L}'_\mathcal{D}\right) &= \left(\mathcal{L}''_\mathcal{A}, \mathcal{L}''_\mathcal{D}\right) \end{aligned}$$

In a disjunction of two formulas, both are evaluated separately and their assignments are concatenated into one list. Assignments with the same variable allocation within this list are combined into one single assignment with a belief vector that is the result of the disjunctive combination of their belief vectors. Duplicates in the list of dependencies from both results are removed. Definition 18 describes this.

Definition 18. *Evaluation of a disjunction*
A disjunction of two formulas $\phi \vee \phi'$ is evaluated as follows:

$$\begin{aligned} eval\left(\phi, \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) &= \left(\mathcal{L}'_\mathcal{A}, \mathcal{L}'_\mathcal{D}\right) \\ eval\left(\phi', \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}\right) &= \left(\mathcal{L}''_\mathcal{A}, \mathcal{L}''_\mathcal{D}\right) \end{aligned}$$

The dependency lists $\mathcal{L}'_{\mathcal{D}}$ and $\mathcal{L}''_{\mathcal{D}}$ are concatenated into one list, doubles are omitted. Equal assignments A and A' in $\mathcal{L}_{\mathcal{A}}$ and $\mathcal{L}'_{\mathcal{A}}$, i.e., those that have the same variables and values, denoted by $\mathcal{EA}$ are combined with $A \vee A'$. Other assignments are denoted by $\mathcal{OA}$

The disjunctive combination of $\mathcal{EA}$s results in $\mathcal{L}'''_{\mathcal{A}}$. Thus the result of a disjunction is:

$$eval\left(\phi \vee \phi', \mathcal{L}_{\mathcal{A}}, \mathcal{L}_{\mathcal{D}}\right) = \left(\mathcal{L}'''_{\mathcal{A}} \cup \mathcal{OA}, \mathcal{L}'_{\mathcal{D}} \cup \mathcal{L}''_{\mathcal{D}}\right) \tag{5.7}$$

A negation is evaluated by applying all available assignments (if any) on the formula within the negation. The formula within the negation is evaluated as described above which results in possibly more than just one assignment. Variables that only occur within the negation are removed from the resulting assignments using the method described in Section 5.1.2.3. All belief vectors of the resulting assignments are then combined by the $\vee$-operator. Thus there is just one resulting belief vector which needs to be negated. If an assignment was applied to the negation, its belief vector is combined with the negated belief vector from the negation. The negation is specified in Definition 19.

Definition 19. *Evaluation of a negation*
Given a list of assignments $\mathcal{L}_{\mathcal{A}}$ and a list of dependencies $\mathcal{L}_{\mathcal{D}}$, $\neg\phi$ is evaluated as follows:

$$eval\left(\phi, \mathcal{L}_{\mathcal{A}}, \mathcal{L}_{\mathcal{D}}\right) = \left(\mathcal{L}'_{\mathcal{A}}, \mathcal{L}'_{\mathcal{D}}\right)$$

Assignments A'_i within $\mathcal{L}'_{\mathcal{A}}$ may have additional variables $Z_1 = w_1, \ldots, Z_n = w_n$:

$$A'_i = \left(\vec{bv'}, [X_1 = v_1, \ldots, X_n = v_n, Z_1 = w_1, \ldots, Z_n = w_n]\right)$$

All variables $Z_1 = w_1, \ldots, Z_n = w_n$ of the negation are removed (see Section 5.1.2.3), which includes calculating the dependencies of $\mathcal{L}'_{\mathcal{D}}$ into the assignments in $\mathcal{L}'_{\mathcal{A}}$. Assignments $A'_1, \ldots, A'_n$ within $\mathcal{L}'_{\mathcal{A}}$ are then combined disjunctively:

$$B'_j = A'_1 \vee \ldots \vee A'_n$$

Only assignment B'_j remains. The belief vector of B'_j

$$B'_j = \left(\vec{bv'}, [X_1 = v_1, \ldots, X_n = v_n]\right)$$

is negated

$$B''_j = \left(\neg\vec{bv'}, [X_1 = v_1, \ldots, X_n = v_n]\right)$$

and extended by A_j:

$$A''_j = B''_j \odot A_j$$

which leads to a new assignment list $\mathcal{L}''_{\mathcal{A}} = [A''_1, \ldots, A''_n]$. Thus the result is:

$$eval\,(\neg\phi, \mathcal{L}_{\mathcal{A}}, \mathcal{L}_{\mathcal{D}}) = \left(\mathcal{L}''_{\mathcal{A}}, \mathcal{L}_{\mathcal{D}}\right)$$

5.1.2.3 Eliminating Variables

Once all predicates of a clause are evaluated and their resulting assignments and dependencies are created, all variables that do not appear in the head of the clause need to be removed. Removing of variables is implemented using a Bayesian Network in which the variables are nodes and dependencies are the conditional probability tables attached to it. We remove those variables for calculating their confidences, specified by masses in their dependencies, into the assignments.

Definition 20. *Bayesian Network from variable dependencies of a clause*
The Bayesian Network is defined by $BN = (N, E)$, where N specifies the nodes and E the edges. Let $P(X, Y_1, \ldots, Y_n)$ be a predicate, X a free (output) variable and $Y_1, \ldots, Y_n$ bound input variables. A value for X depends on the values of $Y_1, \ldots, Y_n$. Thus, every variable is a node within BN: $N = \{X, Y_1, \ldots, Y_n\}$.

$$E \supseteq \{(Y_i, X) \mid Y_i \in N, Y_i \neq X\}$$

A predicate containing only bound variables in its parameters, only creates nodes within N, which are independent.

Definition 21. *Bayesian Network from dependencies*
The Bayesian Network is defined by $BN = (N, E)$, where N specifies the nodes and E the edges. Let $\mathcal{L}_{\mathcal{D}}$ be a list of dependencies $Dep_i = (X_i = v, [Y_{i,1} = w_{i,1}, \ldots, Y_{i,n} = w_{i,n}], m)$, then every $X_i, Y_{i,1}, \ldots, Y_{i,n} \in N$ and every $\{(Y_{i,j}, X_i) \mid Y_{i,j} \neq X_i\} \in E$.

We give a definition for combining an assignment with a dependency, which we need for removing variables.

Definition 22. *The combination of an assignment $A = \left(\vec{bv}, [X_1 = v_1, \ldots, X_n = v_n]\right)$ and a dependency $D = (X_1 = v_1, [X_2 = v_2, \ldots, X_n = v_n], m)$ results in a new assignment A':*

$$A' = \left(\vec{bv} * m, [X_2 = v_2, \ldots, X_n = v_n]\right)$$

Assume a clause that specifies which variables to keep and which to remove looks like this:

$$exampleClause(Y) \leftarrow pred_1(X) \wedge pred_2(X, Y) \wedge pred_3(A) \wedge pred_4(A, Y, Z)$$

In the head of the clause only variable Y occurs which means that the variables X, A, Z need to be removed from the assignments containing all four variables. The network shown in Figure 5.1 is processed from bottom up, until a variable is reached which should be kept. For removing a variable from an assignment, a dependency matching it needs to be combined with the assignment. Therefore, the mass of the matching dependency is multiplied with the belief vector of the assignment with a scalar vector multiplication and the variable is then removed. A dependency is matching if it has the same variables and values as the assignment. Thus dependencies may need to be extended with all its sub-dependencies and independent dependencies in order to build the same frame of discernment.

Figure 5.1 shows an example network, created by variable dependencies.

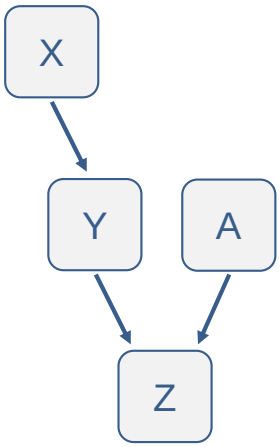


Figure 5.1: Bayesian Network Created by Variable Dependencies

Sub-dependencies are dependencies that arise from the fact that variables within a dependency may depend on variables themselves (A sub-dependency, for example, in Figure 5.1 for variable Y is X). Independent dependencies are those that do not appear within sub-dependencies of a dependency (i.e. in Figure 5.1, A is independent from Y). An extended dependency has to contain exactly the same variables as the assignment in order to match, because they have to be in the same frame of discernment. Definition 25 describes this.

Definition 23. *Extending dependencies:*
Given a list of dependencies $\mathcal{L}_\mathcal{D}$, that span the network $BN = (N, E)$. Further, let E^* be

the transitive closure of E and $\mathcal{L}'_{\mathcal{D}} \subseteq \mathcal{L}_{\mathcal{D}}$ be a list of independent dependencies, containing variables $W_i, \ldots, W_n$. Then $Dep = (X = v, [Y_1, \ldots, Y_n], m) \in \mathcal{L}_{\mathcal{D}}$ can be extended to $Dep' = (X = v, [Y_1, \ldots, Y_n, W_i, \ldots, W_n, Z_i, \ldots, Z_n], m)$ with all Z_i, such that:

$$\forall Y_i : (Z_i, Y_i) \in E^*$$

We define the merging of assignments for joining those, having the same variables and values, but different evidences for true, false, or unknown. We need it for removing variables.

Definition 24. *Merging Assignments*
Let

$$A = \left((m_t, m_f, m_u)^T, [X_1 = v_1, \ldots, X_n = v_n] \right)$$

and

$$A' = \left((m'_t, m'_f, m'_u)^T, [X_1 = v_1, \ldots, X_n = v_n] \right)$$

be two assignments. They can be merged by $\otimes$ to E, if their variables and values are equal.

$$E = A \otimes A' = \left(\vec{bv}, [X_1 = v_1, \ldots, X_n = v_n] \right)$$
$$\vec{bv} = \left(m_t + m'_t, m_f + m'_f, m_u + m'_u \right)^T$$

Definition 25. *Removing variables*
Let $C(\vec{X}) \leftarrow p_1(\vec{Y_1}) \wedge \ldots \wedge p_n(\vec{Y_n})$ be a clause. All variables within $\vec{Y_1}, \ldots, \vec{Y_n}$ that are not in $\vec{X}$ need to be removed. A network out of the variables is build as defined in Definition 20. The network is processed from bottom to top.

Let Z be a variable that needs to be removed and $Dep = (Z = v, [T_1 = w_1, \ldots, T_n = w_n], m)$ a dependency. If list of variables $T_1 = w_1, \ldots, T_n = w_n$ exactly contains all variables within $\vec{Y_1}, \ldots, \vec{Y_n}$, then Z can be removed from

$$Assign = \left(\vec{bv}, [Z = v, T_1 = w_1, \ldots, T_n = w_n] \right)$$

resulting in

$$Assign' = \left(\vec{bv} * m, [T_1 = w_1, \ldots, T_n = w_n] \right)$$

Assignments that are equal afterwards are merged according to Definition 24 and the belief vector of the resulting assignment is normalised to 1 afterwards.

This process repeats until all variables are removed, except for the ones that shall be kept

with their ancestors. The final solution is to be in a compact representation, such that dependencies of Y_i's ancestors are multiplied into Y_i. This is defined in Definition 26.

Definition 26. *Let $D_i = (X = v, [Y_1 = w_1, \ldots, Y_n = w_n], m) \in \mathcal{L}_\mathcal{D}$ be a dependency, such that X is a variable that shall be kept. Further, let $D'_i = (Y_1 = w_1, [], m') \in \mathcal{L}_\mathcal{D}$ a parent of D_i. Y_1 can then be removed from D_i with:*

$$D'' = \big(X = v, [Y_2 = w_2, \ldots, Y_n = w_n], m * m'\big)$$

This is repeated for all D_i, until all ancestors of D_i are aggregated in D_i.

Variables which cannot be removed with the above algorithm, because they appear as dependencies for variables that should be kept, are removed combining the belief vectors of the assignments disjunctively. The final result of the evaluator is a solution containing a list of assignments and a list of dependencies, whereas the assignments only contain the variables of the head of a clause and the dependencies are combined as explained above.

As already mentioned in Section 5.1.2.2, dependencies containing more than one variable in their variable list are omitted in this evaluator, because one cannot directly calculate the masses for single elements out of the combined mass of two or more variables. This is needed when using the *Generalised Bayes Theorem* (Smets [20]) to *turn around* variable dependencies. An example where the masses for single elements need to be calculated is a dependency $P(A, B \mid C, D)$ where A, B depend on C, D, but one wants to calculate $P(D \mid A, B, C)$.

The next section shows how this implementation of a formula evaluator can be used in a tracking module of the CarpeNoctem Behaviour Engine to track activities of team members.

5.2 Tracking Module

This section of the approach describes a possible tracker implementation that uses the evaluator presented in the previous section. Figure 5.2 shows which part of the CarpeNoctem Behaviour Engine needs to be exchanged to be able to evaluate the conditions with the new evaluator, how it is extended to support explicit synchronisation between team members, where a tracking module can be integrated, and how team members can fuse their ball hypotheses based on Dempster-Shafer.

As shown in Figure 5.2, the *ExpressionValidator* needs to be replaced in order to use a

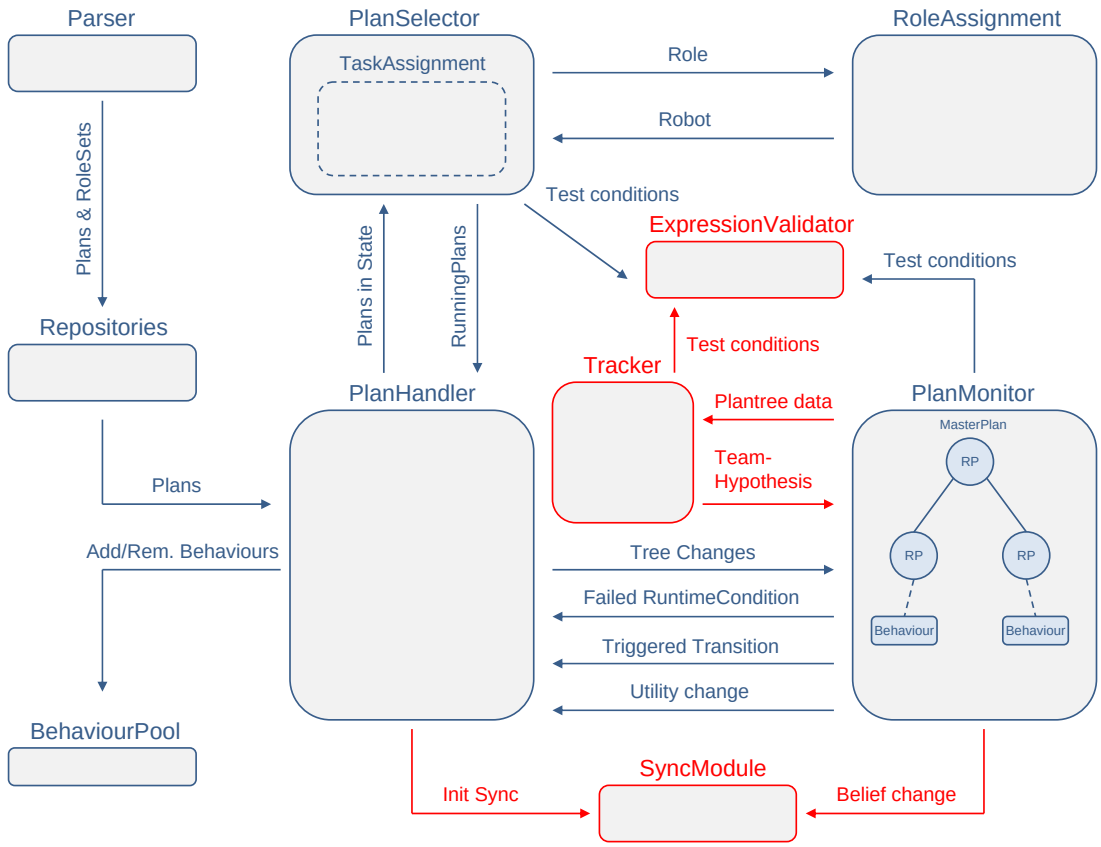


Figure 5.2: CarpeNoctem Behaviour Engine with new Modules (red)

logic programming language for evaluating conditions. Within this thesis such a module was implemented in C#, the language the engine is currently implemented, that is able call Prolog programs through a C interface. Unfortunately, tests showed that the evaluation with the current implementation of the evaluator takes too long to be used in real time within the engine. The tests revealed that an evaluation of a single condition, such as testing if one robot is the closest to the ball with a team of 5 robots, needs $35ms$ on average. The condition is implemented as shown in Formula 5.8.

$$X = robot1 \wedge RobotPos(X, Pos_1) \wedge SharedBall(Ball) \wedge Dist(Pos_1, Ball, D_1) \wedge \neg\,(Robot(Y) \wedge X \neq Y \wedge RobotPos(Y, Pos_2) \wedge Dist(Pos_2, Ball, D_2) \wedge D_2 < D_1) \tag{5.8}$$

Since the main cycle of the Behaviour Engine is triggered every $30ms$ this is way too much.

Considering that the conditions of every robot within the team need to be evaluated and that a robot in average monitors about 10 to 15 transitions in game plans, the evaluation of a condition should take no longer than $0.2ms$.

The following sections describe a theoretical approach for a tracking module and the currently implemented and usable synchronisation of team members and fusion of their ball hypotheses.

5.2.1 Team Hypotheses

Team hypotheses are the central part of the tracking module. A hypothesis for a single agent represents its current actions. Team hypotheses are a combination of such single hypotheses. The combination is described later. The following *node* represents a plan p_1 the team is currently executing:

$$node\left(\vec{bv}_{assign}, \mathrm{In}(a_1, p_1, \tau_1, z_1)_{c_1}, \ldots, \mathrm{In}(a_n, p_1, \tau_n, z_n)_{c_n}\right)$$

Section 3.3 introduced the predicate $\mathrm{In}(a, p, \tau, z)$ to denote that agent a is in state z of plan p and committed to task τ. This predicate is annotated by a belief vector c. Thus, in the node shown above, agent a_1 is committed to task τ_1 and in state z_1 with a belief vector c_1. c_1 only refers to a_1 being in the state z_1, since the confidence for τ_1 is already calculated into the belief vector of the assignment.

Since a plan may have multiple tasks, the *node* contains an assignment describing what team member is allocated to which task. The assignment is annotated with a utility value, used as a confidence. This utility value is in the interval $[-1, 1]$ and can be calculated by the *PlanSelector* module, because this is already done to trigger changing of tasks if a better assignment exists. The utility value reflects how useful a task allocation is for a plan, so it is used as a confidence. For this approach the confidence value is converted into a belief vector in a way that the confidence represents the *true* part and $1 - \mathrm{conf}$ represents the *unknown* part. Since there is no evidence against the assignment, the *false* is 0. If the utility is below zero, no hypothesis for this plan will be created. Formula 5.9 shows such a belief vector:

$$\vec{bv}_{assign} = (conf, 0, 1 - conf)^T \tag{5.9}$$

An initial single agent hypothesis about a robot's states and plans can be obtained from

the plantree information sent to other team members. A belief vector for this single agent hypothesis should express that a team member tells the truth about its state, but also the delay of the network. Thus a value slightly below 1 (ad-hoc chosen: 0.95) is used in this approach for the *true* part and again $1 - true$ is used for the *unknown* part. *False* is 0. Formula 5.10 shows an initial single agent hypothesis for agent a. The plantree is flattened into a vector in a deterministic way.

$$InitHyp(a) = \begin{bmatrix} \mathrm{In}(a, p_1, \tau_1, z_1)_{(0.95,0,0.05)^T} \\ \vdots \\ \mathrm{In}(a, p_n, \tau_n, z_n)_{(0.95,0,0.05)^T} \end{bmatrix} \tag{5.10}$$

Given the description for a *node* and a hypothesis for a single agent, a team hypothesis can be created. The *nodes* described above form a matrix that represents the plan trees of the team members. Such a team hypothesis is denoted by th and shown in Equation 5.11. The indices denote a particular agent out of the team, and particular plan out of its plan tree the agent is in. One can read the matrix from top to bottom as plan trees and from left to right as *nodes* that describe a single plan.

$$th = \begin{bmatrix} node(\vec{bv}_{assign_1}, \mathrm{In}(a_{1,1}, p_{1,1}, \tau_{1,1}, z_{1,1})_{c_{1,1}}, \ldots, \mathrm{In}(a_{1,n}, p_{1,n}, \tau_{1,n}, z_{1,n})_{c_{1,n}}) \\ \vdots \\ node(\vec{bv}_{assign_n}, \mathrm{In}(a_{n,1}, p_{n,1}, \tau_{n,1}, z_{n,1})_{c_{n,1}}, \ldots, \mathrm{In}(a_{n,n}, p_{n,n}, \tau_{n,n}, z_{n,n})_{c_{n,n}}) \end{bmatrix} \tag{5.11}$$

The belief vector for a team hypothesis as a whole, denoted by $\vec{bv}_{th}$, is calculated out of the belief vectors mentioned above. It takes into account the belief vectors of which robot is in which state, denoted by $\vec{bv}_{rob_x}$, the one of the assignment, denoted by $\vec{bv}_{assign}$, and from all nodes, denoted by $\vec{bv}_{node_x}$ within the team hypothesis. All these belief vectors are combined by a conjunction, because all factors have to hold for a team hypothesis. Equation 5.12 shows the relation between the belief vectors.

$$\begin{aligned} \vec{bv}_{node} &= \vec{bv}_{assign} \wedge \vec{bv}_{rob_1} \wedge \ldots \wedge \vec{bv}_{rob_n} \\ \vec{bv}_{th} &= \vec{bv}_{node_1} \wedge \ldots \wedge \vec{bv}_{node_n} \end{aligned} \tag{5.12}$$

Thus, $\vec{bv}_{node}$ describes the belief in the fact that the team of robots executes one plan and $\vec{bv}_{th}$ combines them to reflect the plan trees.

Team hypotheses should be comparable with each other. Because of that a rating over belief vectors is necessary to determine if a team hypothesis is better than the other. Such a rating can be achieved considering the pignistic transformation of Section 2.2 which transfers the belief model into a probability model. In the case of a team hypothesis the *unknown* part is split with equal parts to *true* and *false*. A decision can be taken by comparing the *true* values of the transferred belief vectors.

As described at the beginning of this section, the initial hypothesis can be created using the plan tree information sent by team members. The next section shows how such an initial hypothesis can be updated.

5.2.1.1 Updating Team Hypotheses

Once the tracking module has created the initial team hypothesis it can start tracking the team members' actions. The main indicator for updating team hypotheses are the plantrees sent by team members. An incoming plantree is matched against all tracked hypotheses. If there is a match in the current hypotheses, then the belief vector for that particular robot is replaced with the one for the received plantree information. This replacing process is done for all nodes within the received plantree. If there is no match for the received information, a new team hypothesis is created with the received information.

The advantage in having a tracker and using belief functions is that the frequency for sending out plantree information can possibly be decreased. In order to keep track of the team members, information which was received way back in the past should not be considered as reliable as it was just received. Therefore, Section 2.2 shows that sources of information can be discounted when they are seen as less reliable than before. In every time step, all information regarding which states the team members are in, need to be discounted if there are no new information received from them within this step.

A discounting factor needs to be determined and applied to the belief vectors which are not updated within a tracking step. The discounting factor shifts a portion from the mass in the *true* part of a belief vector towards the *unknown* part. Equation 5.13 shows the discount rule. Since synchronisations use explicit communication and contain information about where in a plan an agent is, those messages can be seen and integrated in the same

way as received plantree information, too.

$$\vec{bv} = (m_t, m_f, m_u) \tag{5.13}$$
$$\text{disc}(\vec{bv}, \alpha) = (m_t - \alpha, m_f, m_u + \alpha) \qquad m_t \geq \alpha$$

Apart from the incoming plantrees to update hypotheses, the *PlanMonitor* of the Behaviourengine calls the *PlanSelector* to test all current assignments for the plans within a robot's plantree. This mechanism needs to be extended to test all assignments within all nodes of the tracked team hypotheses. A test of an assignment evaluates the current task allocation among the team, with respect to the plan's utility function, given the current view of the environment. In case no better assignment for a plan can be determined, the belief vector of the according hypothesis is updated to reflect the newly calculated confidence. If a better assignment is calculated, a new team hypothesis is created with the calculated assignment and its belief vector. The belief vector for the old assignment within the old hypothesis is updated with the calculated confidence. Whenever a change of tasks occurs, the new assignment must be better than the old assignment and the old assignment might even become invalid due to the new environment situation.

Testing whether a better assignment exists not only returns a better assignment for the current plan, but also another plan out of the plan's plan type, together with an assignment for it. In this case, a new team hypothesis is created for the new plan with the newly created assignment and the team members are assumed to be in the initial states as calculated in the assignment. Their belief vectors are initialised with $\vec{bv} = (1, 0, 0)^T$, because the robot that calculated the better assignment believes that its team members have calculated the same assignment and thus changed their states accordingly.

A plan's pre- and runtime-conditions are evaluated by the *PlanSelector* while testing an assignment, too. Thus, if a runtime-condition does not hold anymore, a utility value of -1 would be calculated. This has a direct effect on the team hypothesis containing this assignment, because the belief vector for the assignment will then set to false. Since the assignment's belief vector is calculated into the team hypothesis using a conjunctive rule, it will be false, too.

The core of a possible tracker implementation should not only rely on information sent by other team members, but also determine if they changed their state along a transition to another state. Section 5.1 describes an evaluation implementation for the corresponding conditions which is used in this tracker approach as described in the next section.

5.2.1.2 Monitoring Conditions

In order to track a team member within a plan, an initial hypothesis is necessary to know its current state within the plan. The previous section described how an initial hypothesis can be created from the sent plantree information. Once there is at least one hypothesis for the team member that should be tracked, every outgoing transition of every state in every team hypothesis this team member is supposed to be in, needs to be evaluated. Every such team hypothesis might create new team hypotheses with a possibly new state for the monitored robot. Their belief vector is set to the result of the transition evaluation for the monitored robot to be in the new state. The amount of conditions to evaluate, growth linearly with the amount of robots. The amount of outgoing transitions of a state and the number of plans an agent is in have an even deeper impact on the amount of conditions that need to be checked. Typically in the CarpeNoctem Behaviour Engine, a robot executes four plans in average, thus being in four states with approximately three transitions each. This leads to 12 conditions per robot, per team hypothesis.

A hypothesised state change of a robot also affects the sub nodes of the hypothesis. This is because a state contains plans and the plans of the state that is assumed to be left by the team member contains plans that where tracked, but are left too. If a robot is supposed to stay in its state, the hypothesis containing that information still needs to be updated. The new belief vector for the robot to stay in a state calculates as follows: All outgoing transitions of that state are evaluated, combined in a disjunction, and negated. The previous belief vector is discounted and combined with the calculated belief vector in a conjunctive way. Formula 5.14 describes this.

$$\vec{bv}_{stateNew} = disc\left(\vec{bv}_{stateOld}, \alpha\right) \wedge \neg\left(\vec{bv_{t1}} \vee \ldots \vee \vec{bv_{tn}}\right) \tag{5.14}$$

Section 5.2 already mentions that such an approach needs to be fast because there usually are a lot of transitions to monitor. On the other side, this approach provides a ranking on transitions, i.e., one is able to determine if a robot is more likely to progress along one transition or the other. There is no need for human interaction to tag the transitions with probabilities. Unfortunately, this approach is computationally intensive, because the belief vectors for the conditions need to be calculated, which is a drawback.

5.2.1.3 Hypotheses Pruning

In the prior sections, an explanation is given, how team hypotheses are created and updated. The problem that will arise in practise is the quickly growing number of possible team hypotheses during execution. For a robot to make its decisions, it should select the best team hypothesis to make assumptions about its team members, because it is supposed to be the one that is closest to what really happens. The other hypotheses need to be reduced to a number that can be handled within one run of the engine.

Several approaches can be used to tackle the large number of team hypotheses. The most obvious one would be to take the n best hypotheses and discard all others. The problem with this approach is, that if the tracker tracks the team into a wrong direction for some reason, i.e., it assumes the team to be in another plan and discards the hypothesis with the correct plan, it will stick to this wrong decision until information from the team members arrive, indicating in which plan they are. A smarter algorithm would probably keep some good, some medium, and some bad hypotheses, so it tracks the team in multiple ways and will not get stuck like the first algorithm.

One even better, but more complex algorithm could track multiple options of a plan. This means, if there are multiple hypotheses for the team to be in a certain plan, but with different state information for the team members, more than just one is tracked. This allows the tracker to follow the most likely one, once there is new information about the environment or team status. Another complex algorithm could also investigate the hypotheses and keep those where team members have chosen different plans out of a plan type, so it most likely tracks the correct hypothesis among its tracked hypotheses.

In order to evaluate conditions in the context of team members as described in the previous sections, it is necessary to collect information about their view of the environment and store it in a similar way as the own information about the environment. This is described in the next section.

5.2.1.4 Integrating WorldModel Information

Robots executing the CarpeNoctem Behaviour Engine send out data about their environment in addition to their plantrees. These data contains positions of objects such as the own position, the ball, opponents, etc. With the work of this thesis, these information are stored in a world model that is similar to the own world model, meaning it inherits from the

same base class and one can access information that is available in both world models, in the same way. This is very helpful, because almost all functions that make decisions, take a world model as input data and thus they can also be evaluated in the context of a team member by passing over its world model.

If one would annotate the behaviours and plans of the CarpeNoctem Behaviour Engine with additional constraints, it would be possible to match a team member's world model against these constraints to see if that team member might execute a particular behaviour or plan. For example, such a constraint on an attack behaviour could specify that the robot is the closest to the ball. In literature this is known by the term *plan recognition* or *activity recognition* (Kautz [10]). The tracker approach described in this thesis can easily take information from such a plan recogniser and integrate it into the team hypothesis. Since a hypothesis about which behaviour or plan a robot is executing from a plan recogniser is not very different from the information gathered through a received plantree or synchronisation information. A confidence value could be calculated by the plan recogniser for its hypothesis which then can be transferred into a belief vector that makes it possible to integrate the hypothesis into a team hypothesis.

Apart from the implementation of the evaluator of Section 5.1 and the implementation of team world models just mentioned, there are two other implementations which support the tracker approach explained in Section 5.2. The next sections describe how synchronisations were implemented that can tell the tracker where exactly a robot is within a plan and how the information from the team members about the ball are merged into one *Sharedball* hypothesis.

5.2.2 Synchronisation Points

Synchronisation Points in ALICA are connected to transitions. Agents may only progress along these transitions, once there is at least one agent at every involved transition that believes in the condition attached to it and they mutually believe that this is the case. Therefore, the agents need to communicate explicitly through a synchronisation protocol. The central part of this communication protocol is a message containing the following information:

- AgentID: Id of the agent that belongs to the information in this message
- TransitionID: Id of the transition the agent with AgentID wants to progress along

- ConditionHolds: Boolean value, whether the agent believes that its condition of the transition holds or not
- Ack: Boolean flag that is false if the agent tells other agents its own information and true if it acknowledges information received from others

These messages are exchanged between all members of the team. A synchronisation starts if one agent arrives at a state with an outgoing transition connected to a synchronisation point. This agent sends out its status in a message with the content mentioned above and the *Ack flag* set to *false*. All other agents, involved in this synchronisation, need to acknowledge this message and those that are in a state with an outgoing transition connected to the same synchronisation point will also send out their information and wait for an acknowledgement. Every participating agent builds up a matrix with information about the synchronisation.

AgentID	**TransitionID**	**ConditionHolds**	**Received From**
1	21	true	1,2
2	22	false	1,2,3
3	23	true	1,2,3

Table 5.1: Synchronisation Matrix containing exchanged messages

Let Table 5.1 be the matrix of Agent 1. This agent believes in its condition on the transition it wants to progress along, and one of its team members already has acknowledged that (Agent 2). Agent 2 does not believe in its condition, so it cannot progress along its transition. Because of this and the fact that Agent 1 has not received acknowledgements from all participating agents, the synchronisation is not successful yet. Some time later, the acknowledgement from Agent 3 for the first line may arrive and the belief of Agent 2 about its condition may have changed. Whenever an agent changes its belief about the condition on the transition it wants to progress along, it sends out a new message to its team members to inform them about the change. The new matrix looks like shown in Table 5.2.

AgentID	**TransitionID**	**ConditionHolds**	**Received From**
1	21	true	1,2,3
2	22	true	1,2,3
3	23	true	1,2,3

Table 5.2: Synchronisation Matrix containing exchanged messages for a Complete Synchronisation

In Table 5.2 the synchronisation is complete. All participating agents have acknowledged all rows; they mutually believe that each participating team member believes its condition attached to its transition holds and that the participating team members are in the states connected to the affected transitions. Tests have shown that a synchronisation with a time difference of $16ms$ in average and $2ms$ in the best case, can be achieved this way. The tests were done with PCs that had their clocks synchronised through NTP. The $2ms$ are the best case scenario, because the main loop of the engine, which *moves* the agent to another state after a successful synchronisation, only runs once every $30ms$ and is not synchronised within the team. Thus, the worst case scenario is a synchronisation with a time difference of $30ms$.

These synchronisation points are implemented as a separate module within the CarpeNoctem Behaviour Engine. It is directly connected to the PlanMonitor and PlanHandler as shown in Figure 5.2. A synchronisation is initiated by the PlanHandler as soon as an agent enters a state that is somehow connected to it. The PlanMonitor informs the SyncModul about changes in the belief in the condition of transitions and allows an agent to progress along a transition once a synchronisation is successfully finished.

5.2.3 SharedBall Calculation Based on Dempster-Shafer

For a lot of team decisions in the RoboCup domain, the position of the ball is important. Since the field is too large for a single robot to see the ball, if it is too far away, information about the ball's position needs to be sent around to the team members. All ball hypotheses from the team members need to be merged into a single *SharedBall* hypothesis. Equation 5.8 shows an example condition that relies on the SharedBall, it tests if a certain robot of the team is the closest to the SharedBall.

One simple approach on how to merge the ball hypotheses is to calculate the geometrical centre of them and assume that this is the SharedBall position. The problem with this approach is the robots being not 100% certain where they stand on the field and they are also not 100% sure about the ball position. Depending on how far the ball is away from them or if there are other items looking similar to the ball, the confidence for the seen ball differs.

The approach implemented for this thesis is based on the Dempster-Shafer theory of evidence and takes the position confidences as well as the ball confidences into account while calculating the SharedBall hypothesis. First the ball confidences are discounted with the

position confidences, by multiplying them with each other. This way a hypothesis from a delocalised robot, which sees a ball is weighted lower than one from a robot that knows its position. In the next step, hypotheses with ball positions that are close to each other are assumed to be one. We use the Mahalanobis distance (Mahalanobis [12]) to determine if two positions are close by, because it is able to take the covariance matrices of the positions into account to calculate a more precise centre between the two hypotheses.

To calculate the SharedBall for a team of robots, their ball hypotheses are combined iteratively, meaning the combination of two hypotheses creates a new hypothesis which is then combined with the next one. Each robot may have two ball hypotheses, one for the position where it *sees* the ball and one for *unknown*, reflecting it does not know where the ball is to a certain degree. By combining two hypotheses from two robots, not only the masses of these two hypotheses are taken into account, but also the masses from both robots for unknown. The calculation shown in Equation 5.15 is related to Equation 2.8.

$$
\begin{aligned}
Rob_1 :\ & m(H_1);\ m(U_1) \\
Rob_2 :\ & m(H_2);\ m(U_2) \\
& m(H_{12}) = m(H_1) * m(H_2) + m(H_1) * m(U_2) + m(H_2) * m(U_1) \\
& m(U_{12}) = m(U_1) * m(U_2)
\end{aligned}
\tag{5.15}
$$

This chapter described the implementation of a formula evaluator, written in Prolog, which uses the Dempster-Shafer theory of evidence. It is used in the tracker approach described in Section 5.2 to evaluate conditions and rank them. Further, the implementation of synchronisations, which supports the tracker and the SharedBall, based on the DST is explained. The next chapter evaluates the use of the condition evaluator for a tracking implementation.

6 Evaluation

This chapter shows the evaluation results for the formula evaluator presented in Section 5.1. It shows how it works in a few examples and how a tracker can use it to estimate the behaviour of team members, even if it does not know all information.

6.1 Representing Environment Data in the RoboCup Domain

World model information from a robot is used in the evaluator in the following way: Each sensor information may have a confidence attached to it and usually there is only evidence for a sensor having a specific value, but not against it. Because of this fact, positioning data such as robot positions and ball positions are treated as two positions each.

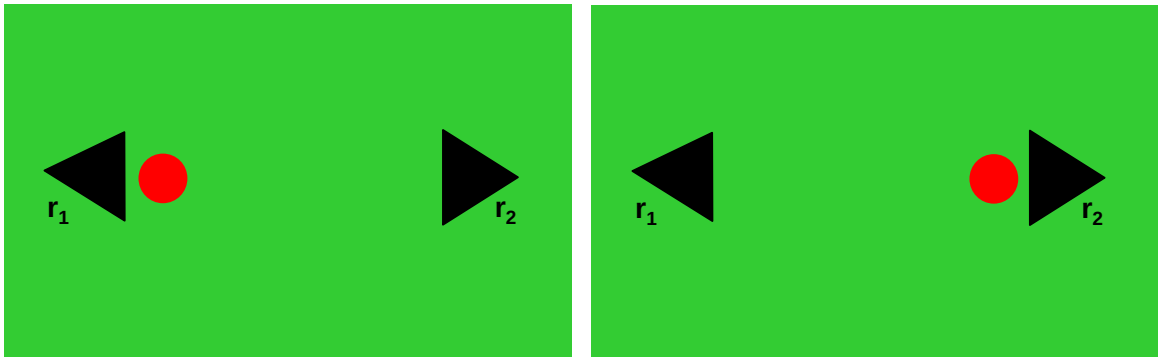
The position of a robot on the field together with the confidence it believes to be at this position is the evidence for the robot to be at this position and thus this confidence is used as a mass for *true*. Since there is no evidence for other positions and against that position in the current CarpeNoctem Behaviour Engine and the sum of masses that covers true, false and unknown is 1, the mass for *unknown* is $1 - m(true)$.

The ball position a robot knows is treated the same way and thus the confidence for the ball is used as the mass for *true* and $1 - m(true)$ is used for *unknown*. The SharedBall and its masses are calculated as explained in Section 5.2.3.

Other data such as situations sent by a RefereeBox computer, data about opponents and more can be annotated with confidences as well and treated as explained above. If there are no sensor information for certain data, only an evidence for *unknown* is used, which then has the mass 1.

6.2 Scenario: Closest Robot to the Ball

Suppose there are two robots within a team, r_1 and r_2. r_1 is in a state that has an outgoing transition with a condition attached, which evaluates to true, if it is the closest robot to the ball. r_2 tries to track r_1 and thus it needs to evaluate the transition for it. Figure 6.1 shows two situations, in Figure 6.1a r_1 is the closest to the ball and in Figure 6.1b it is not. The distance between the two robots is $2m$.



(a) Monitored Robot is the Closest to the Ball (b) Monitored Robot is not the Closest to the Ball

Figure 6.1: Closest Robot to the Ball with 2 Robots

r_2 knows the position of itself and of r_1 as well as the SharedBall position. Using the SharedBall position for such conditions has the advantage of more likely calculating the same results on both robots, because their perceived ball positions might differ. For each of the positions there are two hypotheses, one expressing the real position, and one representing not knowing the position. Equation 6.1 describes the condition.

$$X = r_1 \wedge Robot(X) \wedge RobPos(X, Pos_1) \wedge ShBall(Ball) \wedge Dist(Pos_1, Ball, D_1) \wedge \neg (Robot(Y) \wedge X \neq Y \wedge RobPos(Y, Pos_2) \wedge Dist(Pos_2, Ball, D_2) \wedge D_2 < D_1) \quad (6.1)$$

Given the world model data in Equation 6.2, r_2 calculates the belief vector $\vec{b_1} = (0.78, 0, 0.22)^T$. The pignistic transformation (PT) applied to $\vec{b_1}$ results in a vector with only true and false: $\vec{b_1} = (0.89, 0.11)^T$. This means that the condition is considered to be

true which implies that r_2 assumes r_1 *moves* along this transition to another state.

$$\begin{aligned} \vec{bv}_{SharedBall} &= (0.78, 0, 0.22)^T \\ \vec{bv}_{Pos_1} &= (1.0, 0, 0)^T \\ \vec{bv}_{Pos_2} &= (1.0, 0, 0)^T \end{aligned} \tag{6.2}$$

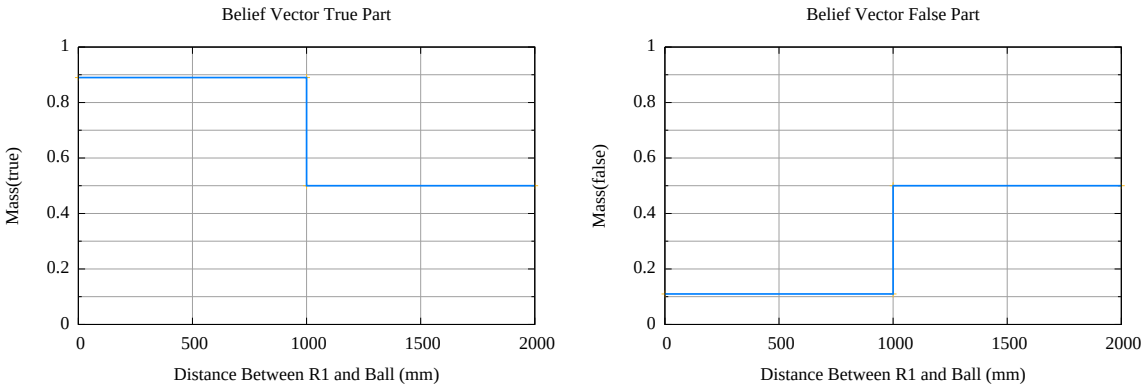
If the ball is closer to r_2[1] as shown in Figure 6.1b the belief vector changes to $\vec{b_2} = (0.39, 0.39, 0.22)^T$. Applying the pignistic transformation to $\vec{b_2}$ leads to $\vec{b_1} = (0.5, 0.5)^T$. This turns out to be wrong, since one would expect $\vec{b_2} = (0, 0.78, 0.22)^T$ and with the pignistic transformation applied: $\vec{b_2} = (0.11, 0.89, 0)^T$. This can be explained by the evaluator not distinguishing between different hypothesis for the same fact and real alternatives. Example 6.3, shows the difference between them which is not (yet) considered in the evaluator. Two robots *bart* and *fransen* want to know if an attacker exists:

$$\begin{aligned} &haveAttacker_1 \leftarrow Robot(bart) \\ &eval(haveAttacker_1, \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}) = ([Assign\left((1,0,0)^T, []\right)], []) \end{aligned} \tag{6.3}$$

$$\begin{aligned} &haveAttacker_2 \leftarrow Robot(X) \wedge X = bart \\ &eval(haveAttacker_2, \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{D}) \\ &= ([Assign\left((1,0,0)^T, [X = bart]\right) \vee Assign\left((0,1,0)^T, [X = fransen]\right)], []) \\ &= ([Assign\left((0.5,0.5,0)^T, []\right)], []) \end{aligned}$$

Figures 6.2a and 6.2b show how the true and false values change when moving the ball.

[1] The SharedBall confidence is assumed to stay the same



(a) Belief Vector True Part

(b) Belief Vector False Part

Figure 6.2: 2 Robots: Belief Vector, True, and False Part

In the scenario explained above, the position confidence is set to 1.0. Assuming the confidences shown in Equation 6.4 for the cases depicted in Figure 6.1 results in the belief vectors: $\vec{b_1} = (0.70395, 0.0, 0.29605)^T$ and $\vec{b_2} = (0.3705, 0.33345, 0.29605)^T$.

$$\begin{aligned} \vec{bv}_{SharedBall} &= (0.78, 0, 0.22)^T \\ \vec{bv}_{Pos_1} &= (0.95, 0, 0.05)^T \\ \vec{bv}_{Pos_2} &= (0.9, 0, 0.1)^T \end{aligned} \tag{6.4}$$

Note that the mass for unknown has increased since the position has some uncertainty in it, too. After applying the pignistic transformation to them, they are: $\vec{b_1} = (0.851975, 0.148025)^T$ and $\vec{b_2} = (0.518525, 0.481475)^T$. Intuitively, this sounds wrong because $\vec{b_2}$ contains more true than false, although the ball obviously is closer to r_2.

Nevertheless the belief vectors show the difference between r_1 being the closest robot to the ball or not. By comparing these belief vectors with those that result when turning around the condition of Equation 6.1, i.e., calculating if r_2 is the closest to the ball, one can see which robot is the closest by comparing the belief vectors. The resulting belief vectors are $\vec{b_3} = (0.351, 0.33345, 0.31555)^T$ and $\vec{b_4} = (0.68445, 0.0, 0.31555)^T$, and with the pignistic transformation applied: $\vec{b_3} = (0.508775, 0.491225)^T$ and $\vec{b_4} = (0.842225, 0.157775)^T$. One can see that in case of Figure 6.1a, $\vec{b_1} = (0.851975, 0.148025)^T$ is more true than $\vec{b_3} = (0.508775, 0.491225)^T$ and in case of Figure 6.1b, $\vec{b_4} = (0.842225, 0.157775)^T$ is more true than $\vec{b_2} = (0.518525, 0.481475)^T$.

We extend this scenario to five robots. The case with uncertain position information is omitted and we investigate only the case with an uncertain SharedBall like in the first scenario. Five situations are shown in Figure 6.3, the positions of the robots are fixed and for each situation, the ball is moved from left to the right.

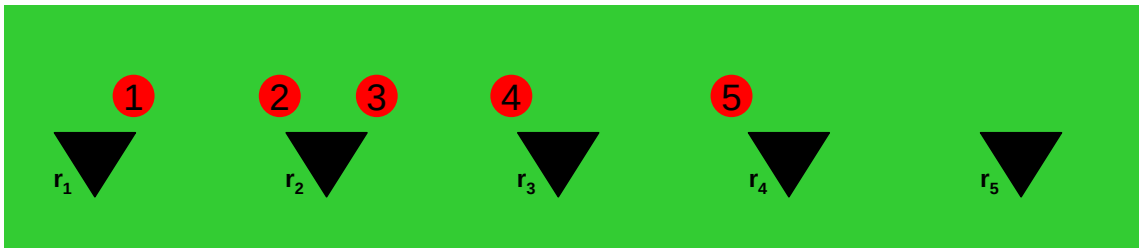


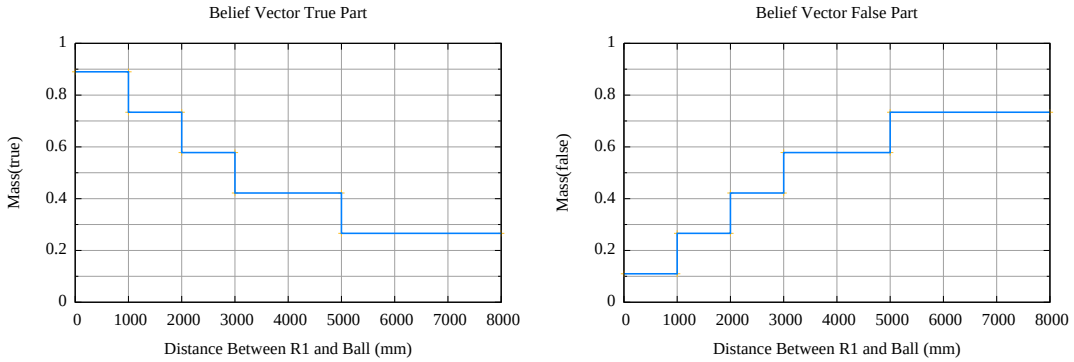
Figure 6.3: 5 Situations with 5 Robots

For reasons of simplicity the robots are positioned with distances of $2m$ to each other and the ball is positioned to either the left or right side of a robot with a distance to it of $50cm$. Again we will calculate if r_1 is the closest robot to the ball with the condition shown in Equation 6.1. In Situation 1, it is the closest to the ball and thus the resulting belief vector is: $\vec{b_1} = (0.78, 0, 0.22)^T$. Situation 2 shows that r_2 is the closest robot and r_1 is the second closest with a belief vector of: $\vec{b_2} = (0.624, 0.156, 0.22)^T$. Moving the ball further away from r_1 so it becomes the third closest robot to the ball as shown in Situation 3, the belief vector is: $\vec{b_3} = (0.468, 0.312, 0.22)^T$. If r_3 is the closest robot to the ball, depicted in Situation 4 and r_1 is the second last, then $\vec{b_4} = (0.312, 468, 0.22)^T$. Finally, if r_1 becomes the robot that is the farthest away from the ball the belief vector is: $\vec{b_5} = (0.156, 0.624, 0.22)^T$. Applying the pignistic transformation to the belief vectors $\vec{b_1}, \ldots, \vec{b_5}$, results in those shown in Equation 6.5.

$$\begin{aligned} \vec{b_1} &= (0.89, 0.11)^T \\ \vec{b_2} &= (0.734, 0.266)^T \\ \vec{b_3} &= (0.578, 0.422)^T \\ \vec{b_4} &= (0.422, 0.578)^T \\ \vec{b_5} &= (0.266, 0.734)^T \end{aligned} \tag{6.5}$$

The belief vectors of Equation 6.5 are shown as diagrams split into the true and false part in Figure 6.4. As one can see, the distance does not matter; it is just the position of the robot within the ranking of who is the closest to the ball. That is why the diagrams contain step patterns, one step, once the position in the ranking of the monitored robot (the one

tested in the condition) changes.



(a) Belief Vector True Part (b) Belief Vector False Part

Figure 6.4: 5 Robots: Belief Vector, True and False Part

6.3 Handling Partly Known Information

Some conditions can only be evaluated partly for team members. This means that these conditions can be evaluated but not as precise as the monitored robot can. Assume a condition for an attacking robot that tries to catch the ball, which tells the robot to *move* from a state with a behaviour for trying to catch the ball into a state where it can start to dribble with the ball. This condition is called: *"have ball"*. The best results for this condition can be reached, if it is evaluated with position and the ball position of that robot that is trying to catch the ball.

Because world model data are sent to other team members, they are also able to evaluate such a condition. However, this cannot be done as precise as by the monitored robot itself, because there is a delay in the network until the information arrives at the team members. Also, as shown in Figure 6.5 a robot does not have exactly the same environment information as the monitored robot. r_2 only has a rough estimation about the ball position ($BP(r_2)$), it is an ellipse covering a wide area, which leads to uncertainty in the calculation of the *"have ball"* condition. In this scenario, the different ball confidence of r_2 is omitted, to keep it simple, although it would further impair the belief vector for the condition. Suppose the discounting factor for the delay in the network is defined to be α and the monitored robot evaluates the condition with a belief vector of $\vec{bv} = (0.9, 0, 0.1)^T$.

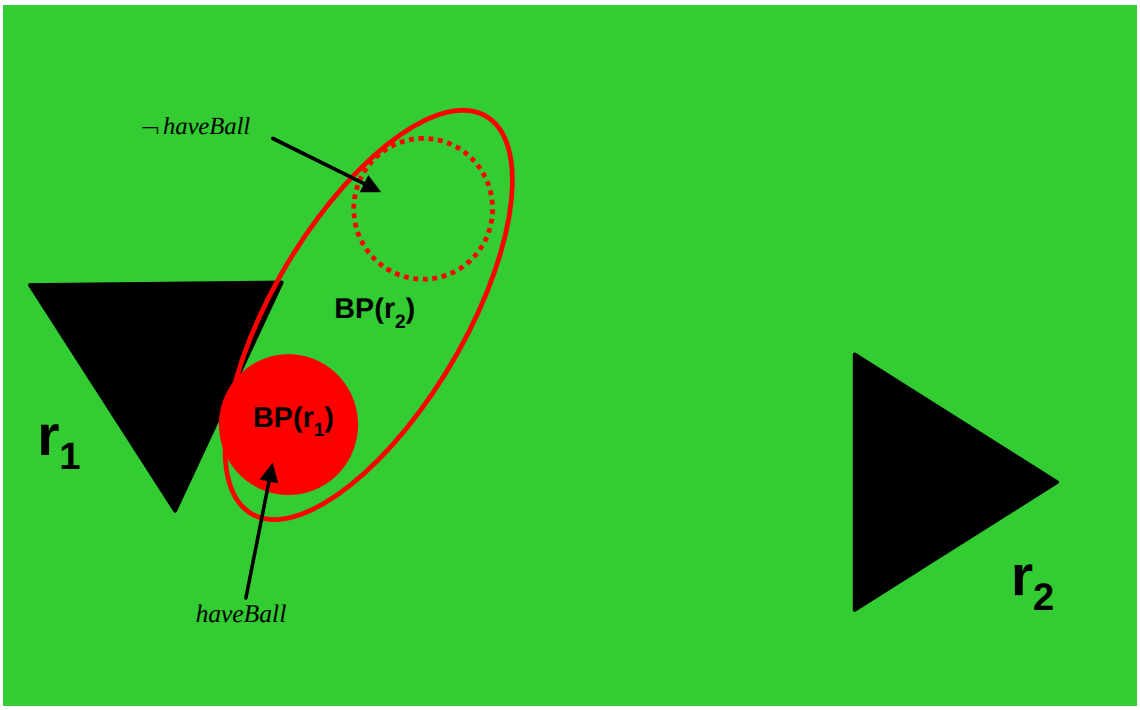


Figure 6.5: Robot r_2 Monitoring if r_1 Has the Ball

Incoming information are discounted with every time step. Thus a team member that evaluates this condition for tracking if the monitored robot changes its state or not, cannot be better than $disc^n\left((0.9, 0, 0.1)^T, \alpha\right) = ((0.9 - \alpha^n), 0, (0.1 + \alpha^n))^T$. n denotes the amount of time steps. Thus, it is still able to evaluate the condition, but with a (higher) mass for unknown. The next section explains how information is treated that is completely unknown.

6.4 Handling Completely Unknown Information

The previous sections of the evaluation showed that a robot is able to evaluate conditions for its team members, even if some of information it needs for evaluation are unknown. This section explains a scenario, in which the necessary information to evaluate a condition might be available or be completely unknown, depending on the situation on the field.

Suppose a team of two robots, r_1 and r_2, playing together and both are able to detect opponents from a distance of at most $6m$. r_1 is the attacking robot that dribbles the ball towards the opponent goal. Its team member r_2 should support him in a way that it blocks

opponents that are in the way between r_1 and the opponent goal. r_2 *moves* between two states, in one state it looks for an opponent to block and once it has found one, it changes to the other state where it blocks the opponent. If the opponent disappears, it *moves* back to the state where it looks for another opponent. r_1 wants to track r_2 and therefore it needs to evaluate its conditions between the two states just mentioned.

We focus on the transition expressing that r_2 has found an opponent. Two scenarios are possible now: First, r_1 is able to see the opponents around r_2 and thus it can evaluate the conditions. The second scenario is, that r_2 is further away from r_1 than $6m$ so r_1 cannot see opponents near r_2. Equation 6.6 shows the world model data available for the first scenario.

$$\begin{aligned} \vec{bv}_{Opp} &= (0.75, 0, 0.25)^T \\ Pos_{Opp} &= (1000mm, 0mm) \\ \vec{bv}_{Pos_2} &= (0.9, 0, 0.1)^T \\ Pos_2 &= (1299mm, 0mm) \end{aligned} \tag{6.6}$$

$\vec{bv}_{Opp}$ specifies the belief vector containing the confidence for the detected opponent. Pos_{Opp} is the detected position of the opponent. $\vec{bv}_{Pos_2}$ describes the belief vector for the position of r_2 and $Pos_2 = (1299mm, 0mm)$ describes its position.

While evaluating the transition with the world model data from Equation 6.6, r_1 is able to calculate the distance between the opponent and r_2, it is $1299mm - 1000mm = 299mm$. To further simplify this scenario, the exact formula is omitted here, but basically it contains the conjunction of $\vec{bv}_{Opp}$ and $\vec{bv}_{Pos_2}$, which in this case results in: $\vec{bv} = (0.675, 0, 0.325)$.

The more interesting scenario is the second one, where r_1 does not know if there is an opponent near r_2. r_1 believes the world model data shown in Equation 6.7. Not knowing where an opponent is, is expressed by a belief vector with the full mass assigned to unknown.

$$\begin{aligned} \vec{bv}_{Opp} &= (0, 0, 1)^T \\ \vec{bv}_{Pos_2} &= (0.9, 0, 0.1)^T \\ Pos_2 &= (1299mm, 0mm) \end{aligned} \tag{6.7}$$

r_1 is not able to calculate a distance between r_2 and an opponent, because it does not know any opponent positions. Thus the distance is treated as unknown and the according belief vector again contains the conjunction of $\vec{bv}_{Opp}$ and $\vec{bv}_{Pos_2}$. This time the result of

the conjunction is $\vec{bv} = (0, 0, 1)$. The two scenarios in this section showed that the tracker approach of this thesis is able to express that certain world model data are completely unknown in an easy way.

6.5 Hypotheses Ranking

Section 5.2.1 describes the core part of the tracker, team hypotheses and how they can be updated. One central question that arises is, if one robot is not only able to tell if a transition of a team member has *fired*[2], but also if a hypothesis containing the assumption that the transition fired is ranked higher than others.

In order to keep the scenario simple, we omit the use of complete team hypotheses and use only one part of them, the hypothesis for one plan which is called *node* in Section 5.2.1. Figure 6.6 shows the plan used in this scenario.

Suppose the team of robots r_1, r_2, and r_3 are executing plan P_A in Figure 6.6. r_1 is in state S_1, r_2 and r_3 are in state S_4 of the plan. Let r_3 be the robot that wants to track only the state changes within this plan of r_1 and r_2. We use the ad-hoc chosen 0.1 as a discount factor to weight down older hypotheses. Further, we assume that r_3 is not able to evaluate the condition on the transition from state S_1 to S_3 for r_1, but it can evaluate the conditions on the other two transitions. It believes the condition on transition from state S_1 to S_2 holds with a belief vector of $\vec{bv}_{T1} = (0.95, 0, 0.05)^T$ in this scenario and r_3 shall have a hypothesis reflecting this, which is ranked higher than its other hypotheses. r_2 further believes the condition on the transition from state $S4$ to $S5$ does not hold with $\vec{bv}_{T2} = (0, 0.9, 0.1)^T$ and the belief vector for the assignment is assumed to stay the same. The initial hypothesis is shown in Equation 6.8.

$$
\begin{aligned}
n_0 = node\left((0.85, 0, 0.15)^T, \mathrm{In}(r_1, P_A, T_1, S_1)_{(0.6,0,0.4)^T}, \mathrm{In}(r_2, P_A, T_2, S_4)_{(0.8,0,0.2)^T}\right) \quad (6.8)\\
\vec{bv}_{n_0} = (0.408, 0, 0.592)^T\\
PT\left(\vec{bv}_{n_0}\right) = (0.704, 0.296)^T
\end{aligned}
$$

Assuming that r_2 stays in its state to further simplify the scenario, there are three possible hypotheses that can be created. Firstly, the one that reflects both robots staying in their

[2] fire in this case means, the condition of the transition was evaluated to true and the corresponding robot moved along it to another state

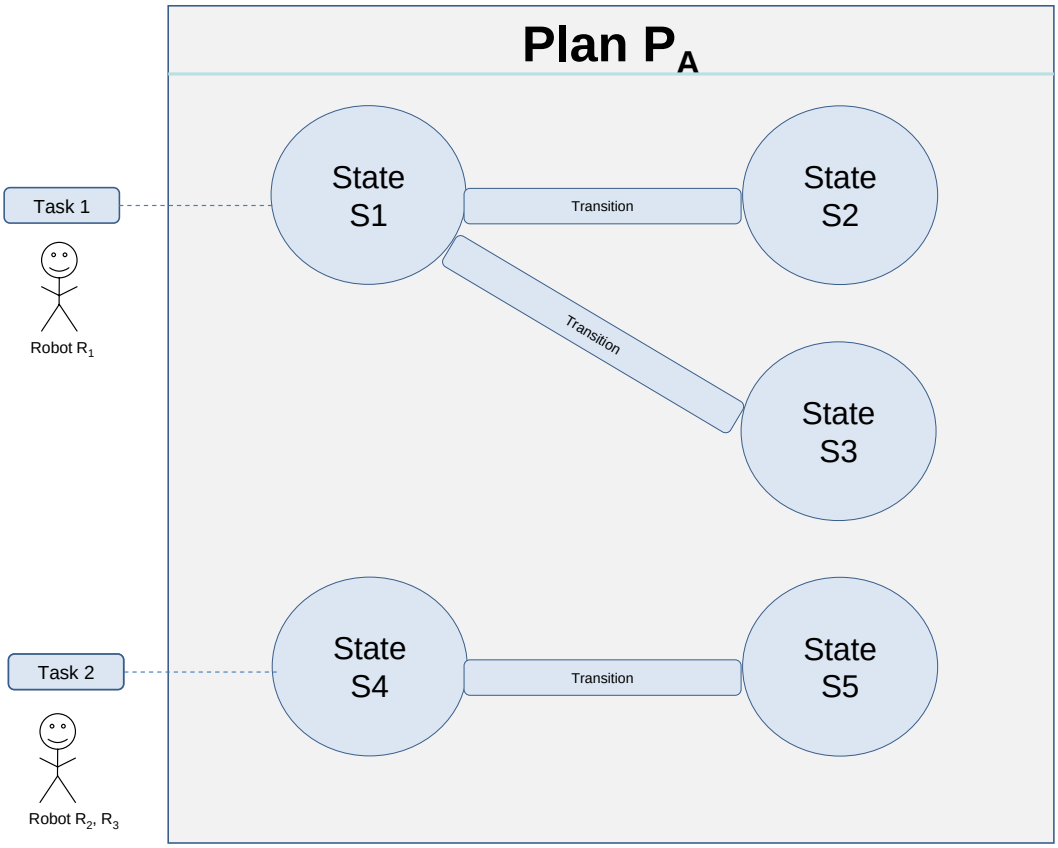


Figure 6.6: Plan for Hypotheses Ranking

states. This means their state hypotheses need to be discounted and combined with the negated conjunction of their outgoing transition evaluations. Equation 6.9 shows these calculations and Equation 6.10 shows the first hypothesis.

$$\begin{aligned} \text{disc}((0.6, 0, 0.4)^T, 0.1) &= (0.5, 0, 0.5)^T \\ \text{disc}((0.8, 0, 0.2)^T, 0.1) &= (0.7, 0, 0.3)^T \\ \neg\left((0.8, 0, 0.2)^T \vee (0, 0, 1)^T\right) \wedge (0.5, 0, 0.5)^T &= (0, 0.8, 0.2)^T \\ \neg\left((0, 0.9, 0.1)^T\right) \wedge (0.7, 0, 0.3)^T &= (0.63, 0, 0.37)^T \end{aligned} \tag{6.9}$$

$$n_1 = node\left((0.85, 0, 0.15)^T, \text{In}(r_1, P_A, T_1, S_1)_{(0,0.8,0.2)^T}, \text{In}(r_2, P_A, T_2, S_4)_{(0.63,0,0.37)^T}\right) \tag{6.10}$$
$$\vec{bv}_{n_1} = (0, 0.8, 0.2)^T$$
$$PT\left(\vec{bv}_{n_1}\right) = (0.1, 0.9)^T$$

In the second hypothesis, r_1 is assumed to have changed its state from S_1 to S_2 via a transition whose condition r_3 cannot evaluate. r_2 stays in its state and its hypothesis is discounted as shown in Equation 6.9. Equation 6.11 shows the second hypothesis.

$$n_2 = node\left((0.85, 0, 0.15)^T, \text{In}(R_1, P_A, T_1, S_1)_{(0,0,1)^T}, \text{In}(R_2, P_A, T_2, S_4)_{(0.63,0,0.37)^T}\right) \tag{6.11}$$
$$\vec{bv}_{n_2} = (0, 0, 1)^T$$
$$PT\left(\vec{bv}_{n_2}\right) = (0.5, 0.5)^T$$

The third hypothesis reflects r_1 changing its state from S_1 to S_3 and again r_2 stays in its state. Thus r_2 needs to be discounted as shown in Equation 6.9 and the belief vector for r_1 is replaced with the one from the transition evaluation. Equation 6.12 shows the third hypothesis.

$$n_3 = node\left((0.85, 0, 0.15)^T, \text{In}(R_1, P_A, T_1, S_1)_{(0.95,0,0.05)^T}, \text{In}(R_2, P_A, T_2, S_4)_{(0.63,0,0.37)^T}\right) \tag{6.12}$$
$$\vec{bv}_{n_3} = (0.508725, 0, 0.491275)^T$$
$$PT\left(\vec{bv}_{n_3}\right) = (0.7543625, 0.2456375)^T$$

One can see that the third hypothesis will be the one that contains the highest probability for true and it reflects the state change of r_1 from S_1 to S_3, which is the case by definition of this scenario.

7 Summary and Future Work

The final chapter of this thesis sums up the work of this thesis and outlines future work that can be done with the tracking approach presented in this thesis.

7.1 Summary

Plan recognition and tracking of agents within multi-agent systems helps to improve the coordination of the agents and can be used to decrease the communication effort between them. Most multi-agent systems rely on the fact that teams of agents need to perform certain tasks together or even synchronously, to achieve their common goal. Unfortunately, robotic multi-agent systems suffer from sensor noise or other imprecise data in real world scenarios. These conditions make it difficult to infer the actions of other agents by observing the environment. Thus, it is also difficult to track them within predefined plans from a plan-library.

This thesis addresses the problem of uncertain and imprecise information while tracking team members. The basis is ALICA, A Language for Interactive Cooperative Agents, in which a human can specify plans that are executed by a team of agents. Each team member sends its internal state to all other team members so they are aware of which plans and behaviours the others are executing. Unreliable communication has an impact on the reception of the internal state messages and thus the coordination of the team members is subject to become worse.

In order to circumvent this problem, a tracking approach was developed in this thesis that is capable of estimating what the team members are doing in case the communication is disturbed. The conditions and utility function of a plan and the conditions on the transitions between states are used to determine the current plans and states of a team member. This is

done by evaluating them from the team member's perspective on the environment. Multiple hypotheses may arise that need to be ranked in order to determine the *best* one, meaning one that suits best the given information about team members and the environment.

Ranking of hypotheses is done using the Dempster-Shafer theory of evidence to combine single agent hypotheses and evaluate a plan's conditions. An evaluator for predicate logic is presented in this thesis (implemented in Prolog) that is able to not only determine whether a formula is true or false, but also tell the amount of truth a formula has under the assumption of certain variable assignment. Furthermore, it is capable of handling information that is not (yet) known and thus a result of the evaluator for a formula is vector of belief masses for true, false and unknown, that can be transferred into a probabilistic model via the pignistic transformation to make decisions.

Unfortunately, tests have revealed that the current evaluator implementation is too slow for real-time usage in a tracker, so there is room for future work. The next section outlines future work that can be done with this thesis.

7.2 Future Work

In this section, an overview should be given about what can be done with the tracking approach presented in this thesis and how it can be extended. Therefore, the following sections describe some ideas on what can be done and what can still be improved.

7.2.1 Representing Alternatives in the Evaluator

As shown in the evaluation of this thesis, the evaluator for formulas does not distinguish between real alternatives and multiple hypotheses for the same fact. Both cases need to be treated differently in terms of combining assignments. Assignments with real alternatives can be combined disjunctively and assignments for multiple hypotheses have to be merged (see Section 5.1.2.3) for merging them.

7.2.2 Implementation of a Tracking Module

In order to use the approach in practise it is necessary to implement it. This thesis describes a basic draft for the implementation of a tracker for a multi-agent system that implements

ALICA in Section 5.2. Figure 5.2 shows how it can be integrated in the CarpeNoctem Behaviour Engine. Given this information, one can implement such a tracking module to decrease the frequency of sending information to team members.

7.2.3 Detecting and Solving Team Play Conflicts

One important fact that can be done with the tracking approach is detecting and solving of coordination conflicts. Such conflicts might be that two or more agents trying to commit to the same task which would violate its cardinality, the team of agents being in different plans of a plan type or that one agent has *moved* along a transition whereas more agents should have moved. In the team hypotheses, these conflicts could be detected and methods can be implemented to react on them.

7.2.4 Plan Recognition

To further improve the team play, one could use plan recognition. Section 5.2.1.4 sketches out that information obtained from the environment can be integrated into the tracking approach presented in this thesis. In order to use plan recognition it would be necessary to annotate plans and behaviours with certain conditions that can be matched against the current view of the environment. Instead of using the world model data that is sent around between team members, the own camera could be used to gain these information and thus further saving communication effort.

7.2.5 RoboCup: Opponent Tracking

Given a working implementation of the tracking approach presented in this thesis, one could create plans for the opponent team and track their members on these plans. This way it is possible to identify attacking and defending robots and try to predict what they will do and how they might react if the own team behaves in a certain way. Having such information about the opponent team helps to adjust the own strategies, matching the game play of the opponent.

Bibliography

[1] Dorit Avrahami-Zilberbrand and Gal A. Kaminka. Fast and complete symbolic plan recognition. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 653–658. Professional Book Center.

[2] Dorit Avrahami-Zilberbrand and Gal A. Kaminka. Towards dynamic tracking of multi-agents teams: An initial report. In *Proceedings of the AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR-07)*, 2007.

[3] Dorit Avrahami-Zilberbrand and Gal A. Kaminka. Incorporating observer biases in keyhole plan recognition (efficiently!). In *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, pages 944–949. AAAI Press, 2007. ISBN 978-1-57735-323-2.

[4] H. A. Blair and V. S. Subrahmanian. Paraconsistent logic programming. *Theor. Comput. Sci.*, 68(2):135–154, 1989. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(89)90126-6.

[5] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1997. ISBN 0121703509.

[6] Didier Dubois and Henri Prade. On the unicity of dempster rule of combination. *International Journal of Intelligent Systems*, 1(2):133–142, 1986.

[7] Stephen S. Intille and Aaron F. Bobick. A framework for recognizing multi-agent action from visual evidence. In *In AAAI-99*, pages 518–525. AAAI Press, 1999.

[8] Finn V. Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001. ISBN 0387952594.

[9] Gal A. Kaminka and Milind Tambe. Robust agent teams via socially-attentive monitoring. *J. Artif. Int. Res.*, 12(1):105–147, 2000. ISSN 1076-9757.

[10] H. A. Kautz. *A formal theory of plan recognition.* PhD thesis, Rochester, NY, USA, 1987.

[11] Neal Lesh and Oren Etzioni. A sound and fast goal recognizer. In *IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence*, pages 1704–1710, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8.

[12] P. C. Mahalanobis. On the generalised distance in statistics. In *Proceedings National Institute of Science, India*, volume 2, pages 49–55, April 1936. URL `http://ir.isical.ac.in/dspace/handle/1/1268`.

[13] Raymond Ng and V. S. Subrahmanian. Probabilistic logic programming. *Inf. Comput.*, 101(2):150–201, 1992. ISSN 0890-5401.

[14] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach.* Prentice Hall, 2nd international edition edition, 2003. ISBN 9788129700414. URL `http://aima.cs.berkeley.edu`.

[15] Glenn Shafer. *A Mathematical Theory of Evidence.* Princeton University Press, Princeton, 1976.

[16] 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.

[17] Christophe SIMON and Philippe WEBER. Bayesian networks implementation of the dempster shafer theory to model reliability uncertainty. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*, pages 788–793, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2567-9. doi: http://dx.doi.org/10.1109/ARES.2006.38.

[18] Hendrik Skubch, Michael Wagner, and Roland Reichle. A language for interactive cooperative agents. Technical report, University of Kassel, 2009.

[19] Philippe Smets. The nature of the unnormalized beliefs encountered in the transferable belief model. In *Proceedings of the eighth conference on Uncertainty in Artificial Intelligence*, pages 292–297, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1-55860-258-5.

[20] Philippe Smets. Belief functions: The disjunctive rule of combination and the generalized bayesian theorem. *International Journal of Approximate Reasoning*, 9(1):1–35, August 1993.

[21] Philippe Smets and Robert Kennes. The transferable belief model. *Artif. Intell.*, 66 (2):191–234, 1994.

[22] V. S. Subrahmanian. On the semantics of quantitative logic programs. In *SLP*, pages 173–182, 1987.

[23] V. S. Subrahmanian. Paraconsistent disjunctive deductive databases. *Theor. Comput. Sci.*, 93(1):115–141, 1992. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(92)90214-Z.

[24] Gita Sukthankar and Katia Sycara. Simultaneous team assignment and behavior recognition from spatio-temporal agent traces. In *AAAI'06: Proceedings of the 21st national conference on Artificial intelligence*, pages 716–721. AAAI Press, 2006. ISBN 978-1-57735-281-5.

[25] Gita Sukthankar and Katia Sycara. Hypothesis pruning and ranking for large plan recognition problems. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 998–1003. AAAI Press, 2008. ISBN 978-1-57735-368-3.

[26] Milind Tambe. Recursive agent and agent-group tracking in a real-time dynamic environment. In *ICMAS*, pages 368–375, 1995.

[27] Milind Tambe. Tracking dynamic team activity. In *AAAI/IAAI, Vol. 1*, pages 80–87, 1996.

[28] Milind Tambe and Paul S. Rosenbloom. Resc: an approach for real-time, dynamic agent tracking. In *IJCAI'95: Proceedings of the 14th international joint conference on Artificial intelligence*, pages 103–110, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8, 978-1-558-60363-9.

[29] Stefan Triller. A cooperative behaviour model for autonomous robots in dynamic domains, January 2009.

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