Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-91-11

Distributed

Truth Maintenance

Thilo C. Horstmann

May 1991

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Distributed Truth Maintenance

Thilo C. Horstmann

DFKI-D-91-11

Distributed Truth Maintenance[1]

Thilo C. Horstmann

German Research Center for AI (DFKI)
Project KIK
P.O. Box 2080
W-6750 Kaiserslautern
Germany

e-mail: *horstman@dfki.uni-kl.de*

May 1991

Abstract

Distributed AI systems are intended to fill the gap between classical AI and distributed computer science. Such networks of different problem solvers are required for naturally distributed problems, and for tasks which exhaust the resource of an individual node. To guarantee a certain degree of consistency in a distributed AI system, it is necessary to inspect the beliefs of both single nodes and the whole net. This task is performed by Distributed Truth Maintenance Systems. Based on classical TMS theories, distributed truth maintenance extends the conventional case to incorporate reason maintenance in DAI scenarios.

[1]This work was done in Project KIK at the German Research Center for Artificial Intelligence (DFKI). Project KIK is a collaborative effort between the DFKI and Siemens AG.

Contents

List of Figures

Chapter 1

Introduction

Recent research in the field of Distributed Artificial Intelligence (DAI) has led to a broad variety of applications characterized by autonomous, loosely connected problem solving nodes. Each single node, or *agent*, is capable of individual task processing and able to coordinate its actions in combination with those of other agents in the net. DAI applications span a large field ranging from cooperating expert systems, distributed planning and control to human computer cooperative work. In order to establish a domain independent theory of interacting autonomous agents, current DAI research focuses on defining an abstract agent model, which allows the formalization of cooperation strategies and multi agent reasoning mechanisms.

The requirements of multi agent reasoning algorithms are manifold. In most cases, it is *not* desirable to constrain the autonomy of agents by building a 'superstrate reasoner' managing all inferences or rules of a set of different agents. The reasons are discussed fully in [DLC89]. Instead, we want the agents to be able to reason autonomously; in particular, a single agent must deal with beliefs, which have probably been created in a complex cooperation process.

This requirement is best fulfilled by providing an agent with a Distributed Truth Maintenance System (DTMS)[1]. Based on classical TMS theories, distributed truth maintenance extends the conventional case to make reason maintenance suitable for multi agent scenarios. A DTMS has to represent and manage inferences and rules of interacting agents in a way that ensures a specified degree of consistency. The various degrees of consistency will be defined in this paper. Furthermore, other modules of an agent should be able to use information stored by the DTMS. For instance, a problem solving unit may avoid recomputation or a cooperation process might be based on the current context of the consistent knowledge base.

We start off by presenting a Truth Maintenance System designed for backward reasoning

[1] We use the term Truth Maintenance System instead of the perhaps more appropriate term Reason Maintenance System or Belief Revision System. This is done for historical reasons.

systems. We shall show that the properties offered by a TMS for forward reasoning systems, can also be used by backward reasoning systems. Beyond, the amalgamation of the *incrementality* and *selectivity* of a justification based TMS with the properties of backward reasoning allows elegant and efficient programming techniques in a first-order logic representation.

The basic key features of the DTMS are summarized below:

- maintenance of a consistent state of beliefs. Because we record data dependencies checking consistency involves little recomputation when the knowledge base is modified.
- data dependencies are recorded in the Horn subset of first-order predicate logic instead of propositional logic.
- explicit representation of proofs allows for easier generation of explanations.
- interface for exchanging beliefs, data and proofs among agents.
- meta level predicates aiding the design of clearly specified autonomous agents. In addition, it simplifies the classification of goals into those upon which reason maintenance should be performed and those which remain static.
- the DTMS is designed as a generalization of a TMS. As a result, the application domain is not restricted to the field of DAI.

In these terms, the results of this work may be divided into two main chapters. Chapter 2 introduces the basic terminology and discusses TMS techniques tailored for backward chaining resulting in a *Backward Reasoning Truth Maintenance System (BRTMS)*. This chapter is not specific to the area of DAI and should be useful to readers interested in areas such as TMS, meta logic programming and backward reasoning. In Chapter 3, the BRTMS of Chapter 2 is extended to the distributed case. We define central terms concerning multi agent reasoning and illustrate the DTMS algorithm.

Chapter 2

A JTMS for Backward Reasoning Systems

Das Erst wär so, das Zweite so,
Und drum das Dritt und Vierte so;
Und wenn das Erst und Zweit nicht wär,
Das Dritt und Viert wär nimmermehr.

The First was so, the Second so,
Ergo the Third and Fourth ensued;
But given no First nor Second, no
Third, yea, nor Fourth had been or could.

— J. W. v. GOETHE, FAUST I

In the last decade, the desire for non-monotonic reasoning systems and more efficient search strategies in problem solving algorithms, generated a considerable amount of research in the field of truth maintenance systems (TMS). We distinguish two main categories of TMS. *Justification* Based TMSs (JTMS) as introduced by J. Doyle in 1979 and *Assumption* Based TMSs (ATMS), presented first in 1985 by J. de Kleer.

A TMS works as an independent module connected to a problem solving unit in a knowledge based system. Based on a set of dependencies, the JTMS assigns belief statuses to data representing the current context of the database. The truth maintenance procedure guarantees consistency and a well founded basis for beliefs in the face of a changing set of dependencies. It keeps track of all inferences made[1], so that recomputation of inferences can be avoided. Additionally, most TMSs allow the problem solver to reason in a non-monotonic way (e.g., "infer the sensor is ok unless there is evidence to the contrary") and to deal with contradictions. Contradiction resolution is performed by a procedure called *dependency-directed backtracking* which can be implemented on a JTMS by identifying and adding absent justifications. Thus, a JTMS ensures a contradiction-free database.

In contrast, the ATMS computes, for each datum, all contexts in which it is valid. A context

[1]The BRTMS presented here allows one to declare upon which inferences truth maintenance should be performed.

is defined by a set of consistent assumptions. De Kleer's ATMS [dK86][2] "precomputes all answers to all possible queries". Queries must also be posed with a subset of all possible premises. In addition to the JTMS functionality, the ATMS computes the minimal sets of assumptions necessary to prove a given formula.

We do not want to discuss ATMS vs. JTMS in detail; there are a lot of publications dealing with this issue. The reader is referred to [McA90], where a good introduction to Truth Maintenance is presented. However, we do provide reasons for using a JTMS as the basis for truth maintenance in distributed scenarios in Chapter 3.

2.1 Forward versus Backward Chaining

Former TMSs have been designed for use with incremental forward reasoning systems. In a forward reasoning system, each inference step produces new conclusions from antecedent data, which can be passed to the TMS. In contrast, in a backward reasoning system each inference step does not produce new conclusions, rather new conditions for the goal[3]. To make conclusions, we have to wait until the reasoning process is complete. In these terms, the problem solver would have to keep track its inferences, in order to transmit appropriate data to a classical TMS.

However, the designer of the problem solver should not have to think about how to represent inferences. Our system relieves the system designer from this task, all inference control is done by meta logic predicates in the BRTMS. We shall come back to the differences between these kinds of reasoning systems in Section 2.2.5 and 2.5.

2.2 Basic Terminology

2.2.1 BRTMS Architecture

Figure 2.1 shows the general architecture of the Prolog-based BRTMS. The BRTMS *Meta Level* includes meta logic predicates, user defined justifications, the current state of beliefs, the BRTMS *Kernel* system predicates and user defined static predicates. The meta level controls the evaluation of all goals, performs the bookkeeping of results and defines the BRTMS interface while the kernel defines low level predicates: predicates, which might be evaluated through a meta call, but whose proof is not significant for the BRTMS bookkeeping mechanism (see also Section 3.3.2).

[2] De Kleer presented a modified version in 1990 [dK90b]. This one allows negative literals in queries, the dependencies are stored as Boolean Constraints instead of material implications.

[3] Throughout this paper we use the terminology of logic programming as introduced in [Llo84].

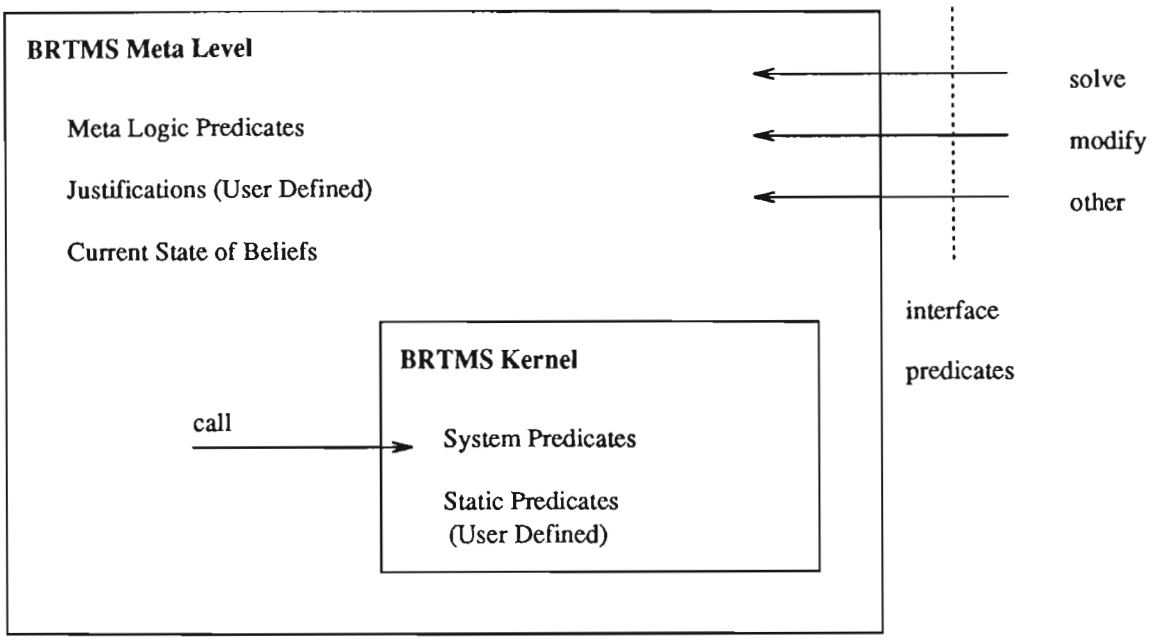


Figure 2.1: BRTMS Architecture.

The meta logic predicate `dtms_solve/5`, that we discuss in more detail later, plays a central role in the meta level. The definition of `dtms_solve/5` realizes a modification of a standard Prolog meta interpreter. At first sight, this interpreter takes as an argument a Prolog query g and tries to find a proof for g in accordance with clauses of the kernel, the meta level and with the current set of beliefs. In the course of doing this, all data dependencies are stored or updated as necessary. One important feature concerning this meta concept should be mentioned at this point: The designer of BRTMS applications is freed from creating data dependencies, all dependencies are implicitly defined by justifications.

Justifications are defined in the meta level. These are dynamic program clauses defining the atomic formulas (or *atoms*), on which truth maintenance will be performed. In former JTMSs, justifications are –in a different form– the only kind of rules. But we will see when considering BRTMS applications that the combination of a TMS with a Prolog problem solver increases the TMS functionality by allowing for system predicates. As mentioned before, these predicates are also evaluated by `dtms_solve/5`. Furthermore, we will see that there is a whole class of predicates that should be evaluated in the same manner as system predicates. These are predicates that are never be altered such as `member/3` or `append/3`. Obviously, there is no point in performing truth maintenance on those predicates. Because of these reasons, we define the BRTMS Kernel. In the kernel all predicates of the meta level are invisible, but the meta level can evaluate predicates defined here. The proof tree of the result of a kernel call will not be stored. In other words, you may regard the kernel may be regarded as the 'static true world' and the meta level as the 'dynamic changing world'.

Modifications to justifications must be done through the meta-level predicates. This means in particular, the kernel predicates may not assert or retract justifications. Violating this principle would result in a undefined behavior of the BRTMS, because each modification of the justification database invokes the meta level `dtms_solve/5` predicate and possibly a call to the initial kernel predicate again. This architecture allows two different ways of designing BRTMS applications. We will refer to it in Section 2.5.

2.2.2 Major Data Structures

Querying the BRTMS invokes the interpretation of a finite set of Horn clauses. As mentioned in the last section, we divide this set into two disjunct sets: the set of (dynamic) justifications $\mathcal{J}$,[4] and the set of static and system predicates $\mathcal{S}$. When called, the BRTMS will create or modify *beliefs*. Informally, a belief is an atomic formula to which is assigned four fields:

- *status:* one of the symbols `in` or `out`
- *constraint:* a first-order formula in conjunctive normal form
- *support:* a list of atomic formulas
- *consequences:* a list of atomic formulas

The *status* field designates belief (if `in`) or lack of belief (if `out`). The formula in *constrain* can be regarded as the reason for assigning this status. Furthermore, the fields *support* and *consequences* denotes the dependencies of the atom according to the current set of beliefs. That is, *consequences* represents those beliefs which might have to be recomputed if the status changes. In the other direction, the elements of *support* are those beliefs, upon which the status of the atom is dependent.

We define these notions precisely in the following:

Definition 1 (State) *Let $\mathcal{A}$ be a finite set of positive literals, $\mathcal{L}(\mathcal{A})$ the set of all subsets of elements of $\mathcal{A}$ and $\mathcal{F}(\mathcal{A})$ the set of all formulas constructed of elements of $\mathcal{A}$. A* state Ψ *is a 4-tupel $\Psi = (\lambda, \mu, \nu, \xi)$ such that*

(i) $\lambda : \mathcal{A} \rightarrow \{\texttt{in}, \texttt{out}\}$

(ii) $\mu : \mathcal{A} \rightarrow \mathcal{F}(\mathcal{A})$

(iii) $\nu : \mathcal{A} \rightarrow \mathcal{L}(\mathcal{A})$

[4] A justification must be a clause with non empty body. A justification with the symbol *true* as its body is called a *premise* justification.

(iv) $\xi : \mathcal{A} \rightarrow \mathcal{L}(\mathcal{A})$

Definition 2 (Instance) *Let* $\theta = \{v_1/t_1, \cdots, v_n/t_n\}$ *be a substitution and* j *be a justification. Then* j_θ *is an* instance *of* j, *if each occurrence of variable* v_i *in* j *is simultaneously replaced by the term* $t_i (i = 1, \cdots, n)$.

Definition 3 *Let* $\Psi = (\lambda, \mu, \nu, \xi)$ *be a state and let* j_θ *be an instance of a justification with head* h_θ *and body* b_θ. *Furthermore, let* $\mathcal{P}$ *be a set of program clauses. Then* j_θ *is*

(i) valid *(w.r.t.* Ψ*), if* θ *is a correct answer substitution for* $\mathcal{P} \cup \{b\}$[5]. *In this case, for each positive literal* p *of* $\mathrm{b}_\theta, \lambda(p) =$ `in` *and for each positive counterpart of a negative literal* n *of* $\mathrm{j}_\theta, \lambda(n) =$ `out`. *We say,* j_θ *justifies* h_θ.

(ii) invalid *(w.r.t.* Ψ*), if* θ *is not a correct answer substitution for* $\mathcal{P} \cup \{b\}$. *In this case, there is either a positive literal* l *of* b_θ *with* $\lambda(l) =$ `out`, *or a positive counterpart of a negative literal* l *of* b_θ *with* $\lambda(l) =$ `in`. *We say,* l *invalidates* j_θ.

Example: Let $\mathcal{A} = \{a(2), b(2), c(2), d(2)\}$ with $\lambda(a(2)) = \lambda(d(2)) =$ `in` and $\lambda(b(2)) = \lambda(c(2)) =$ `out`. Furthermore, we have three justifications j1: $d(X) \leftarrow a(X), \neg b(X)$, j2: $d(X) \leftarrow a(X), b(X)$ and j3: $c(X) \leftarrow d(X)$. Then the instance of $j1_{\{X/2\}}$ is valid, but $b(2)$ invalidates $j2_{\{X/2\}}$.

Definition 3 generalizes the notion of propositional justifications to the first order case. In these terms, a justification in a first-order logic TMS represents the set of all instances of the justification. To introduce the central term *consistency*, we need some further definitions.

Definition 4 (Inval) *Let* $\Psi = (\lambda, \mu, \nu, \xi)$ *be a state and let* $j_1, \cdots, j_n$ *be a set of justifications with the same predicate* a *in the head. If each justification* j_i *is invalidated by a* b_i, *we will denote the set* $\{b_1, \cdots, b_n\}$ *by* inval (a).

Definition 5 (Definition of a Justification) *Let* p *be a predicate and* $\mathcal{J}$ *be a set of justifications. The* definition of a justification (written *def (p)*) *is the disjunction of all bodies of clauses of* $\mathcal{J}$ *with the same predicate* p *in the head.*

Definition 6 *Let* c *be a conjunction of atomic formulas* $a_1, \cdots, a_n$. *Then [c] denotes the set of atoms* $a_1, \cdots, a_n$.

Definition 7 *Let* $\Psi = (\lambda, \mu, \nu, \xi)$ *be a state.* con (a) *is the set of atoms whose members are those atoms* $c \in \mathcal{A}$, *such that* a *is a member of* $\nu(c)$.

[5] That is, $\forall(b\theta)$ is a logical consequence of $\mathcal{P}$.

Definition 8 (Consistency) *Let $\Psi = (\lambda, \mu, \nu, \xi)$ be a state, and $\mathcal{P} = \mathcal{J} \cup \mathcal{S}$ be a union of disjunct sets of program clauses. Ψ is* consistent, *if the following conditions hold:*

(i) if $\lambda(a_\theta) =$ `in`*, then either*

- *(a) there is a $j \in \mathcal{J}$ such that j_θ justifies a_θ. In this case, $\mu = b_\theta, \nu = [b_\theta], \xi = con(a_\theta)$ or*
- *(b) θ is a correct answer substitution of $\mathcal{S} \cup \{a\}$. In this case, $\mu = system$[6], $\nu = [system], \xi = con(a_\theta)$.*

(ii) if $\lambda(a_\theta) =$ `out`*, then θ is not a correct answer substitution of $\mathcal{P} \cup \{a\}$. If there is a definition for a, then $\mu = cnf(not(def(a)))$[7], $\nu = inval(a)$, $\xi = con(a_\theta)$. In each other case $\mu = system$, $\nu = [system]$, $\xi = con(a_\theta)$.*

(iii) there is no sequence $(a_0, \cdots, a_n)$ of elements of $\mathcal{A}$, such that $a_0 = a_n$ and for $i = 1, \cdots, n, \lambda(a_i) =$ `in` *and a_{i-1} is in $\mu(a_i)$.*

Definition 8 implies some notable points. Condition (iii) prohibits circularities in the support of `in` beliefs in order to establish a well founded set of atomic formulas (see also Definition 14). Furthermore, a consistent state of beliefs guarantees a correct assignment of logical states to atomic formulas *and* a correct linkage of all beliefs in accordance to their logical dependencies.

Example: The state in the example of Definition 3 is *inconsistent*: the belief $c(2)$ is `out` but the instance $j3_{\{X/2\}}$ is valid. Thus, condition (ii) of Definition 8 is violated.

In order to represent a consistent state of beliefs, the BRTMS stores for an atom `datum` the corresponding values of λ, μ, ν, ξ in the arguments `status`, `constraint`, `support` and `consequences` of `dtms_node/6`[8] (Figure 2.2).

```
dtms_node (datum, status, constraint, support, consequences, rule_id)
```

Figure 2.2: Definition of `dtms_node/6`

We say, `datum` is in, or `datum` is *believed*, if the status field of `datum` has the value `in`.

Note, the symbol `true` can occur in the support field of `datum` in two cases: `datum` is `in` and justified with a premise justification, or `datum` is `out` and the justifications matching with

[6] In these terms, the symbol *system* denotes the support for a belief that is inferred from kernel predicates

[7] cnf denotes the conjunctive normal form of a given formula

[8] The abstract mathematical object *set* is represented with the Prolog object *list*.

datum contain no further subgoals. For instance, the justification $p \leftarrow fail$, is invalidated by the symbol fail.[9] Because $cnf(not(def(p))) = true$, the support field of a belief p would be the symbol true.

In addition, the argument rule_id denotes a unique identifier of the justification supporting a believed datum. The general representation of justifications in the BRTMS is shown in Figure 2.3.

```
dtms_rule (justification, rule_id)
```

Figure 2.3: Definition of dtms_rule/2

The heads of justifications define the atoms on which truth maintenance is performed. This is a noteable difference from classical TMSs, in which these atoms have to be declared explicitly. We will see this in more detail later. Figure 2.4 defines a family relationship with justifications.

```
dtms_rule ((grandchild (X, Y) :- granddaughter (X, Y)), gc1).
dtms_rule ((grandchild (X, Y) :- grandson (X, Y)), gc2).
dtms_rule ((granddaughter (X, Z) :- daughter (X, Y),
          child (Y, Z)), gd).
dtms_rule ((grandson (X, Z) :- son (X, Y), child (Y, Z)), gs).
dtms_rule ((child (X, Y) :- daughter (X, Y)), ch1).
dtms_rule ((child (X, Y) :- son (X, Y)), ch2).
dtms_rule ((son (jake, bill) :- true), s1).
dtms_rule ((son (bill, scott) :- true), s2).
```

Figure 2.4: Family Tree.

The following figures give an example for a consistent state of beliefs that is created by the BRTMS with the use of the justifications above. Note that in this example there is no predicate defined in the BRTMS kernel, all beliefs are inferred by use of justifications only.

In our family example the query

```
?- dtms_solve (grandchild(X,Y), Status, Support, Mode, yes).
```

would yield the creation of the beliefs shown in Figure 2.5 and 2.6: Dependencies are represented as in [Doy79]: Arrows represent justifications pointing to the justified belief. Positive signed arcs represent positive literals. Note that the first order representation of dependencies requires regarding justifications as a whole.

[9] The beliefs true and fail are in and out but not explicitly represented in the BRTMS.

```
dtms_node (daughter(_g59, _g61), out, true, [true], [], _g55).
dtms_node (child(jake, bill), in, son(jake, bill),
          [son(jake, bill)], [], ch2).
dtms_node (son(bill, scott), in, true, [true],
          [child(bill, scott)], s2).
dtms_node (son(jake, bill), in, true, [true],
          [grandson(jake, scott), child(jake, bill)], s1).
dtms_node (child(bill, scott), in, son(bill, scott),
          [son(bill, scott)], [grandson(jake, scott)], ch2).
dtms_node (grandchild(jake, scott), in, grandson(jake, scott),
          [grandson(jake, scott)], [], gc2).
dtms_node (grandson(jake, scott), in,
          (son(jake, bill) , child(bill, scott)),
          [son(jake, bill), child(bill, scott)],
          [grandchild(jake, scott)], gs).
```

Figure 2.5: Family Tree (cont.).

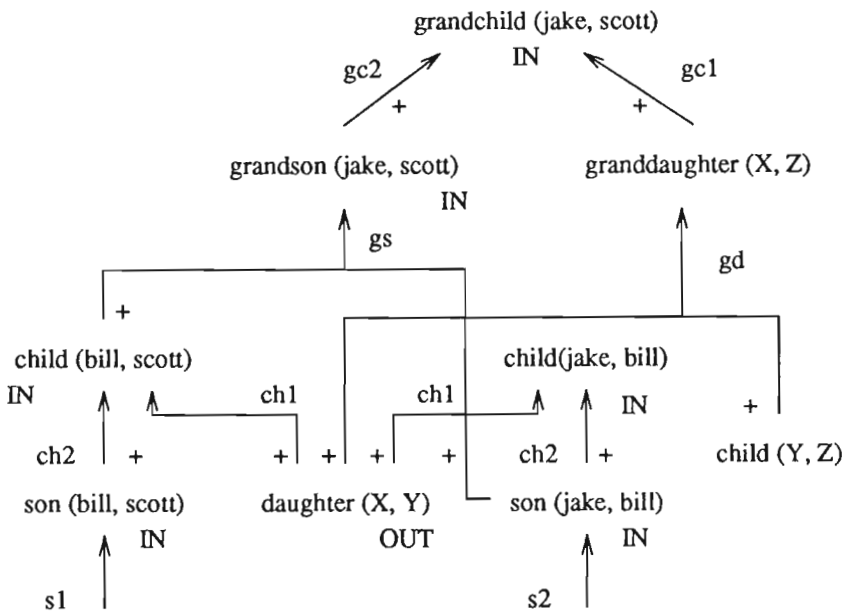


Figure 2.6: Vizualisation of Dependencies.

We might interpret the last belief of Figure 2.5 as follows: `grandson (jake, scott)` is believed, because both `son (jake, bill)` and `child (bill, scott)` are believed. `grandson (jake, scott)` is responsible for the current status of `grandchild (jake, scott)`. The selected justification proving `grandchild (jake, scott)` is the justification with rule id `gs`.

2.2.3 The Meta Level Predicates

As mentioned before, the BRTMS is based on a Prolog meta interpreter. This interpreter is defined by the predicate `dtms_solve/5`, which we now discuss. Note that `dtms_solve/5` works destructively, because it creates and modifies instances of the data structure `dtms_node/6`.

Its main tasks are to

- compute the states of atoms according to justifications, static predicates and system predicates.
- create or modify instances of the data structure `dtms_node/6` to maintain a consistent database and to link the beliefs according to their logical dependencies

Given a consistent state of beliefs, calling `dtms_solve/5` will create all those beliefs, such that the (instantiated) literals of the first argument of `dtms_solve/5` will occur in the data base and the newly created state of beliefs is consistent[10].

Figure 2.7 shows the recursive definition of `dtms_solve/5`.

The first two clauses split the query and recurs on the first literal and the remainder of the first argument. `status` and `constraint` are constructed in accordance with the state of the subgoals.

Clauses three and four concern negated literals. A negated atom is `in` if the positive counterpart of the atom is `out` and vice versa (see Definition 3).

The next clause of `dtms_solve/5` returns the status `in` and the corresponding `constraint` of an atom g, if there is a database entry for g. Access to the belief database can be avoided, if the argument `Mode` is bound to the symbol `tms`.

Clause six realizes the kernel call. If there is a clause in the kernel whose head matches with g, then the predicate `monotonic_goal/1` succeeds. In this case, the kernel clause is called. This fulfills b of Definition 8.i.

If a kernel call fails, the BRTMS tries to prove the atom g with justifications. If it succeeds, the beliefs will be either created or modified in accordance to the previous data dependencies

[10]The BRTMS is correct, but not complete (see Section 2.3.1).

```
dtms_solve ((G1, G2), Status, (Con1, Con2), Mode, Rem) :- !,
       dtms_solve (G1, St1, Con1, Mode, Rem),
       dtms_solve (G2, St2, Con2, Mode, Rem), !,
       status_and (St1, St2, Status).
dtms_solve ((G1; G2), Status, (Con1; Con2), Mode, Rem) :- !,
       dtms_solve (G1, St1, Con1, Mode, Rem),
       dtms_solve (G2, St2, Con2, Mode, Rem), !,
       status_or (St1, St2, Status).
dtms_solve ((not Goal), out, Constraint, Mode, Rem) :-
       dtms_solve (Goal, in, Constraint, Mode, Rem).
dtms_solve ((not Goal), in, Constraint, Mode, Rem) :- !,
       dtms_solve (Goal, out, Constraint, Mode, Rem).
dtms_solve (Goal, in, Constraint, Mode, Rem) :-
       Mode \== tms,
       clause (dtms_node (Goal, in, Constraint, _, _, _)).
dtms_solve (Goal, in, system, Mode, Rem) :-
       monotonic_goal (Goal), !,
       call (Goal).
dtms_solve (Goal, in, Body, Mode, Rem) :-
       clause (dtms_rule ((Goal :- Body), Rule)),
       dtms_solve (Body, in, Constraint, Mode, Rem),
       update_node (Goal, in, Body, Rule, Mode, Rem).
dtms_solve (Goal, out, Constraint, Mode, Rem) :-
       Mode \== tms,
       clause (dtms_node (Goal, out, Constraint, _, _, _)).
dtms_solve (Goal, out, system, Mode, Rem) :-
       monotonic_goal (Goal), !,
       not call (Goal).
dtms_solve (Goal, out, Constraint, Mode, Rem) :- !,
       get_all_clauses (Goal, Body),
       cnf (not Body, Constraint), !
       dtms_solve (Body, out, Con, Mode, Rem),
       update_node (Goal, out, Constraint, _, Mode, Rem).
```

Figure 2.7: Definition of `dtms_solve/5`.

and those encountered in the call to the predicate `update_node/6`[11]. We do not present the implementation of `update_node/6` here in more detail, because of its routine nature. `update_node/6` realizes the linkage of each belief w.r.t. the current state, i.e. it computes and establish the values of `consequences` and `support` (see also Definition 8).

Furthermore, if a belief n exists and changes its status, `update_node/6` calls the predicate `dtms_solve/5` with all consequences of n as the first argument. This *downstream propagation* of the changes in the status field of a belief guarantees a consistent state of beliefs if they do not contain any circularities in their dependencies (see Section 2.4). We also use `update_node/6` to implement predicates modifying the justification database.

The last 3 clauses of `dtms_solve/5` are invoked if the status of the atom g is `out`. In the last one, we built the conjunctive normal form of all bodies of justifications that match the current atom. We already mentioned the aspect in Definition 8.

2.2.4 Further Definitions

In this section we briefly present the basic TMS terminology that is necessary for understanding the following chapters. Most of it has been introduced for classical TMS by [Doy79].

Definition 9 *Let $\Psi = (\lambda, \mu, \nu, \xi)$ be a state and $a \in \mathcal{A}$ a belief. Then*

(a) the supporting beliefs *of* a *are the set of literals of $\nu(a)$.*

(b) if $\lambda(a) =$ `in` *we also call the supporting beliefs of* a antecedents.

(c) a foundation *of* a *is recursively defined to be either* a *or a foundation of the antecedents of* a.

(d) an ancestor *of* a *is recursively defined to be either* a *or an ancestor of the supporting beliefs of* a.

(e) the consequences *of* a *are the set of literals of $\xi(a)$.*[12]

(f) if $\lambda(a) =$ `in` *we also call a consequence of* a believed consequence.

(g) a believed repercussion *of* a *is recursively defined to be either* a *or a believed repercussion of the believed consequences of* a.

[11] Creation or modification of beliefs can be suppressed, if the symbol `rem` (remember) is bound to no.

[12] In Doyle's terminology, this set is called *affected consequences*.

(h) *a* repercussion *of* a *is recursively defined to be either* a *or a repercussion of the consequences of* a.

2.2.5 Comparison with Classical JTMS Data Structures

To bring the introduction of the basic data structures to a close, we compare the BRTMS data structures with those of the Doyle system. Doyle's justifications are lists of the form (in-list out-list). A justification is valid, if each node of its in-list is labeled `in`, and each node of its out-list is `out`. A node is a data structure containing the fields justifications, consequences, status and support. A node is `in`, if it has at least one valid justification in its justifications list, otherwise it is `out`. For instance, Figure 2.8 shows a network of dependencies in the Doyle system. q has a premise justification, because its supporting jus-

$j1 : (()(\neg p))$
$j2 : (()())$
$j3 : ((p)())$

p :	$\neg p$:	q :
$justifications : (j1)$	$justifications : ()$	$justifications : (j2 \quad j3)$
$consequences : ()$	$consequences : (p)$	$consequences : ()$
$status : in$	$status : out$	$status : in$
$support : (\neg p)$	$support : ()$	$support : ()$

Figure 2.8: Dependencies in Doyle's TMS

tification j2 has an empty in and an empty out list. Doyle calls the symbol p an *assumption*, because p is in, unless there is a valid justification for ¬p. Actually, there are two separate nodes to represent p and ¬p with the following behavior: If we add a valid justification to the justification list of the node ¬ p, ¬p will go in and p will be out. But if we retract the justification j1 of p in the initial situation above, both p *and* ¬p would go out. Note that Doyle's truth maintenance algorithm assigns values to the status and support of each node. It does not create nodes and does not alter consequences and justifications of nodes.

The corresponding representation in the BRTMS is:

```
dtms_rule(p :- not '¬p', j1).
dtms_rule(q :- true, j2).
dtms_rule(q :- p, j3).
```

Querying the BRTMS yield the following beliefs:

```
dtms_node(p, in, not '¬p', ['¬p'], [], j1).
dtms_node('¬p', out, true, [true], [p], _g55).
dtms_node(q, in, true, [true], [], j2).
```

The creation of beliefs with all fields automatically assigned is a important difference from Doyle's system. The user needs only to create justifications, the BRTMS computes all data dependencies and stores them into instances of `dtms_node/6`. The particularly implementation is efficient as a result of the Prolog unification algorithm.

2.3 Discussion

The design of a TMS as a variant of a Prolog meta interpreter yields some notable points which are discussed in this section.

2.3.1 Completeness of the BRTMS

The current version of the BRTMS is logically correct but not complete. This means, if the `dtms_solve/5` query terminates, the created state of beliefs is consistent w.r.t Definition 8. On the other hand, there are justification sets admitting the creation of a consistent state of beliefs, which are not detected by our algorithm. Furthermore, if no consistent state exists, it runs into an infinite loop instead of reporting failure. Consider the examples of Figure 2.9. They both have in common circular data dependencies. Example (a) may be represented with the following rules: `dtms_rule ((q :- p), j1)`, `dtms_rule ((p :- q), j2)`, `dtms_rule ((p :- true), j3)` while the odd loop in example (b) might be represented with `dtms_rule ((q :- p), j1)`, `dtms_rule ((p :- not q), j2)`. Clearly, in example (a) a `dtms_solve/5` query should create a consistent state with both `p` and `q` labeled `in` while a query in example (b) should report failure. In both cases, non-terminating execution is due to the Prolog depth first search algorithm. A complete algorithm would have to avoid visiting the same state twice; for instance by collecting the already visited states in a list. But this might be an expensive task. Imagine the following justification set:

```
dtms_rule ((a0 :- a1), j0).
dtms_rule ((a1 :- a2), j1).
                ⋮
dtms_rule ((an :- true), jn).
```

To label the belief *a0*, at first the labels of the beliefs $a1 \cdots an$ have to be computed. Our algorithm performs the labeling of *a0* in linear time in the size of the justification set.

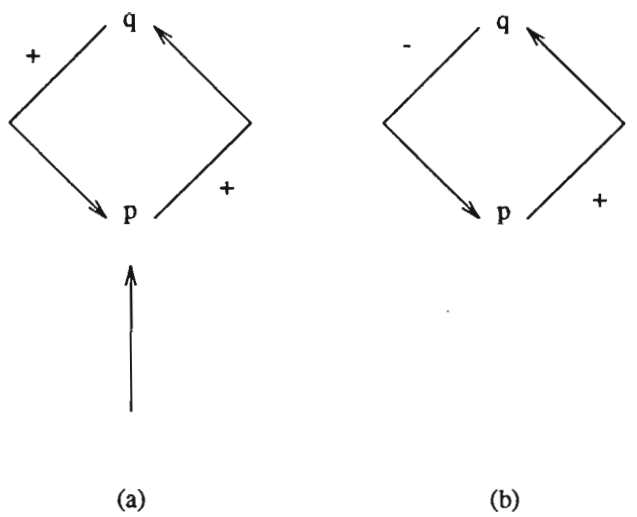


Figure 2.9: Loops in BRTMS.

However, a complete algorithm, like Boolean entailment, is coNP complete and no efficient algorithm can be expected [McA90].

2.3.2 The Lemma Generation Problem

The BRTMS stores, for a given query, the set of beliefs that are necessary for the created state to be consistent. The same query again does not require recomputation. But difficulties occur if we want more than one solution. Consider again the family example. A first query

```
?- dtms_solve (grandchild(X,Y), Status, Support, Mode, yes).
```

yields the creation of the beliefs of Figure 2.4 and the bindings

```
X = jake, Y = scott, Status = in, Support = grandson(jake, scott)
```

by leaving `Mode` uninstantiated. Doing the same query later on, will results the same bindings by retrieving the information of the database entry. This is perfectly done. But, initiating backtracking at this point, in order to collect further solutions, results in computing the same answer again. Certainly, this behavior is contrary to our intention of getting an answer like 'no more solution'. The explanation of this misbehaviour is: The first answer results from the database access, the second from the deduction of the goal with the use of the internal clauses.

This is the *lemma generation problem*[13] [Sou90]: The rediscovery of previously found solutions before finding any new ones. Certainly, the lemma generation problem is not to be solved by modifying our meta interpreter so that `dtms_solve/5` fails, if a query is solved by using the internal program clauses and if a corresponding database entry already exists.[14] However, this would result in the correct behavior but also in redundant computation.

A possible solution of the lemma generation problem consists of explicitly representation the search tree for a given query [Sou90]. Each node of this tree represents a goal and is related to a list of program clauses matching the goal. In general, such a search tree contains exhausted nodes, that is nodes that cannot contribute further solutions, and nodes marked open. The latter are related to the remaining possibilities. To examine further solutions, a proof procedure can only find the remaining alternatives in the open search space. For a more detailed discussion see [Sou90].

In fact, the current version of the BRTMS does not implement a solution of to the lemma generation problem. But the meta level approach of [Sou90] might be incorporated in our system in a natural way. This is a subject of further research.

2.4 Altering the Justification Database

In order to allow the problem solver to reason non-monotonically, TMSs allow for assertion or to retraction of justifications. In general, adding or retracting justifications disrupt a consistent state of inferences made previously, so that the truth maintenance procedure is invoked to reestablish a consistent state.

In former TMSs, each node is associated with a list of justifications. Thus, altering the justifications means adding or retracting a justification to the justification list of one single

[13] A *lemma* corresponds to our term *belief*

[14] This could be realized by modifying clause 7 and 10 of `dtms_solve/5` as follows:

```
dtms_solve (Goal, in, Body, Mode, Rem) :-
       clause (dtms_rule ((Goal :- Body), Rule)),
       dtms_solve (Body, in, Constraint, Mode, Rem),
test for a database entry
       update_node (Goal, in, Body, Rule, Mode, Rem).
                                          :
dtms_solve (Goal, out, Constraint, Mode, Rem) :- !,
       get_all_clauses (Goal, Body),
       cnf (not Body, Constraint), !
       dtms_solve (Body, out, Con, Mode, Rem),
test for a database entry
       update_node (Goal, out, Constraint, _, Mode, Rem).
```

node. But in the general case one must retract a justification globally, that is not only in *one*, but in *each* node concerning this justification. In order to do this, the TMS has either to search for all nodes depending on j or to associate justifications with its corresponding nodes in some way.

The Prolog unification algorithm allows establishing such a bidirectional association of beliefs and justifications without explicit declarations. Thus, if the justification database implies is altered, the BRTMS will alter *all* beliefs depending on this justification if necessary.

In the next both paragraphs, we assume a initial state $\Psi = (\lambda, \mu, \nu, \xi)$ to be consistent. We discuss modifications of the justification database via the predicates `assert_justification`, and `retract_justification/1`.

2.4.1 Adding Justifications

The predicate `assert_justification (clause, id)` needs in its first argument the justification and in the second the unique rule id. The algorithm is simple:

collect all database entries of `dtms_node/6` whose atomic formula matches the head of the asserted justification and whose status is `out`.

call `dtms_solve/5` with each of these atoms.

The definition of the predicate `update_node/6` (see Section 2.2.3) performs the downstream propagation of changing belief statuses. This involves all depending repercussions of a belief in the relabeling process. As seen, this algorithm works well for non circular justification sets. But a complete algorithm needs to be more sophisticated. Consider the example in Figure 2.10 [Rus85]. Before adding the justification represented by the dotted arcs, the shown consistent state might have been constructed. To reestablish consistency, the relabeling procedures have to be able to propagate the changes *upstream* as well. Relabeling upstream means, involving all depending foundations of a newly justified belief in the relabel process. In the example, these are the beliefs r and s. The only known complete relabel algorithm [Rus85] makes s `in` and all other beliefs `out`.

2.4.2 Retracting Justifications

Retracting a justification works in a similar way. `retract_justification/1` requires the rule id of the relevant justification and performs the following:

- collect all those database entries, such that the corresponding belief is `in` and is justified with the retracted justification.

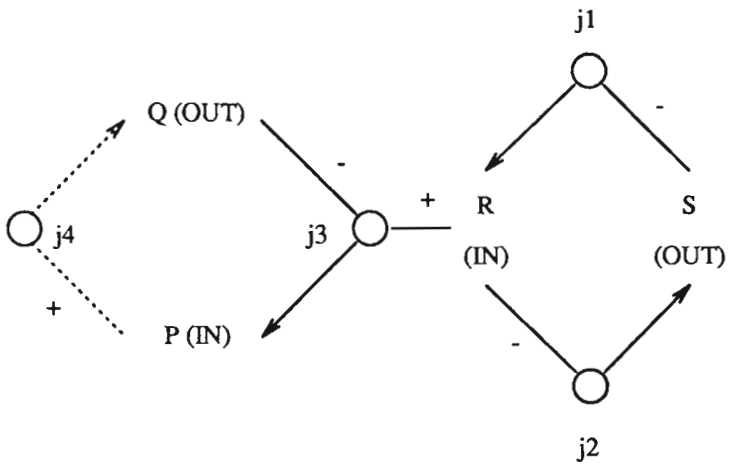


Figure 2.10: Adding a Justification.

- call `dtms_solve/5` with each of these atoms.

Thus, the query

```
?- retract_justification (s2).
```

will cause the beliefs of Figure 2.11 to be created in the family example.

2.5 Designing Applications with the BRTMS

This section discusses how to use the BRTMS in real applications. We present two possibilities that are characterized by the way in which the BRTMS is connected to the problem solver. In the *segregated* approach the BRTMS is connected with the problem solver as an independent module while in the *synergetic* approach the problem solver also uses the BRTMS inference mechanisms.

2.5.1 The Segregated Approach

Usually, truth maintenance systems are designed as segregate units connected to the problem solver (Figure 2.12). This arrangement is appropriate for forward reasoning systems. Starting with ground facts, the forward inference engine of the problem solver deduces, after each inference step, ground conclusions that can be easily transmitted to the TMS. In

```
dtms_node(child(jake, bill), in, son(jake, bill),
        [son(jake, bill)], [], ch2).
dtms_node(son(jake, bill), in, true, [true],
        [child(jake, bill)], s1).
dtms_node(son(bill, scott), out, true, [true],
        [child(bill, scott)], _g55).
dtms_node(daughter(bill, scott), out, true, [true],
        [child(bill, scott)], _g55).
dtms_node(child(bill, scott), out,
        (not son(bill, scott) , not daughter(bill, scott)),
        [son(bill, scott), daughter(bill, scott)],
        [grandson(jake, scott)], _g55).
dtms_node(daughter(jake, bill), out, true, [true],
        [granddaughter(jake, scott)], _g55).
dtms_node(grandson(jake, scott), out,
        not son(jake, bill) ; not child(bill, scott),
        [child(bill, scott)], [grandchild(jake, scott)], _g55).
dtms_node(granddaughter(jake, scott), out,
        not daughter(jake, bill) ; not child(bill, scott),
        [daughter(jake, bill)], [grandchild(jake, scott)], _g55).
dtms_node(grandchild(jake, scott), out,
        (not grandson(jake, scott) , not granddaughter(jake, scott)),
        [grandson(jake, scott), granddaughter(jake, scott)], [], _g55).
```

Figure 2.11: Family Tree Revisited.

contrast, a backward reasoning system deduces, after each inference step, new conditions for a goal that are inappropriate to transmit to the TMS. If a backward reasoning system like Prolog uses a TMS in this manner, the transmission of conclusions to the TMS has to be done *after* the reasoning process is complete. Thus, the designer of the problem solver has to realize an algorithm that generates an explicit representation of the proof for a given goal. This representation of a proof can be transmitted to the TMS. A rough algorithm schema is (all clauses in the problem solver might be identified by an unique id):

- collect for a given query q the ids $id1 \cdots idn$ of those rules used by the interpreter to deduce the query.
- create with these ids the implication $q' \leftarrow id1\& \cdots \& idn$ where q' denotes a substitution of q.
- transmit this implication to the TMS via `add_justification`. Provided that all ids are present in the TMS as premises, the TMS creates the belief q' with status to be `in`.

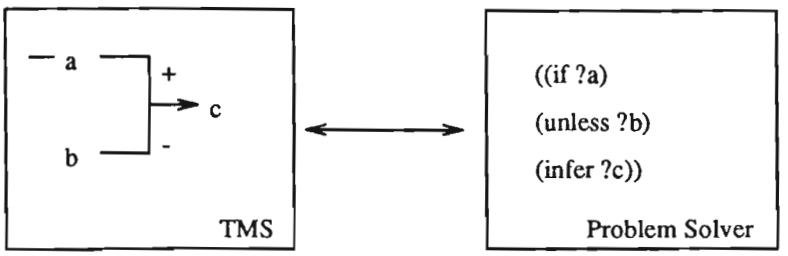


Figure 2.12: TMS Connected to Problem Solver

In these terms, a TMS designed as a segregate module might represent its data dependencies in propositional logic formulas. However, a backward reasoning problem solver using such a TMSs, does have to control its inference process to supply the TMS with appropriate data. Our BRTMS relieves the programmer of this task: He can integrate most of the problem solver functionality into the BRTMS.

2.5.2 The Synergetic Approach

The synergetic approach of designing BRTMS applications is the result of using the BRTMS inference mechanisms. As mentioned in Section 2.2.1, we represent in the kernel the 'static true word' and in the meta level the 'changing dynamic world'. Thus, designing a typical BRTMS application starts by classifying goals either in dynamic or static. A good advice might be the remark of Goethe's Mephisto: *Ei, was ich weiß, das brauch ich nicht zu glauben* (Ai, what I know I have no need of believing). In other words: Selecting goals on which truth maintenance is performed is a trade-off between avoiding unnecessary recomputation and overwhelming the BRTMS with too much information.

Figure 2.13 shows a fragment of an expert system that aids in selling cars. In the BRTMS kernel, we place the physical car data, and relations depending on physical car data. For instance, the facts `space/3` represent the space inside a car. In the meta level there are heuristics assigning values to those physical data and beliefs representing the current set of preferred cars. In contrast to the physical data of the kernel, it is possible to modify these heuristics later on.

In a connected module we can model algorithms, which modify the justification database of the BRTMS. In the example, we compare in `buy_car/3` the car suggested by the BRTMS with experiences other persons have already had. By taking account of negative experiences, we create a new 'prefer heuristic' in order to find another car.

We see, a designer does not have to think about how to pass justifications to the TMS or how to establish the correct linkage of beliefs. All that is done by the BRTMS. Certainly, the BRTMS does not save the application designer from creating appropriate heuristics.

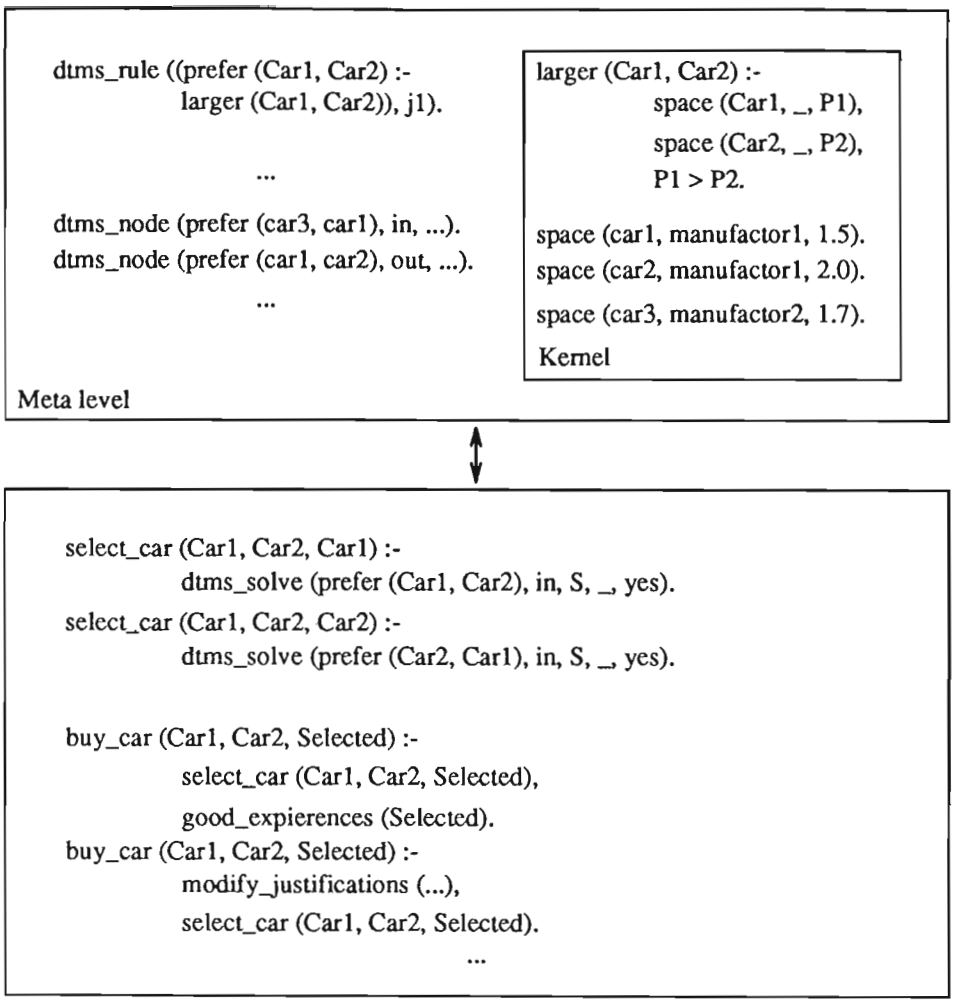


Figure 2.13: Example for a Synergetic BRTMS Application

But he can be sure while establishing or changing heuristics that all inferences ever made are consistent w.r.t. the current heuristics.

Chapter 3

Extension of the BRTMS to Distributed Truth Maintenance

Uns trennt das Schicksal, unsere Herzen bleiben einig!
Fate will us separate, but united our hearts remain!

— F. SCHILLER, WALLENSTEIN'S TOD

In the current chapter we present a way to establish reasoning among interacting autonomous agents. We define an abstract terminology that clearly specifies basic terms of multi agent reasoning. The central term is that of *proof consistency*. In contrast to previous attempts, this definition of consistency in multi agent scenarios is characterized by exchanging beliefs as well as exchanging reasons for the beliefs. We can see this in everyday life: When debating an issue, we do not want to know *what* somebody claims, but, in addition *why* he claims it. Furthermore, we usually agree with somebody only if we agree with him in his conclusion *and* in the foundations of his conclusion.

In these terms, interacting agents, which exchange beliefs along with the corresponding foundations, reason much more flexibly than agents which only transmit the results of inferences. If an agent later invalidates the foundation of an acquired belief, it might reconsult the agent from which the belief was originally acquired.

But, we do not want to overwhelm an agent with too much information by transmitting complex traces of inferences between agents. We will show that it is sufficient to transmit only a special representation of proofs and not the whole proof structure. In addition, the designer of multi agent scenarios can specify a *level of consensus*, which defines the knowledge upon which the agents will agree all the time. This feature can greatly improve efficiency in multi agent reasoning.

We do not force the agents to agree on all information - our notion of a proof consistent state allows agents to be partially inconsistent with one another. That is, agents might have

different viewpoints of certain beliefs. If two agents reason together to solve the query "Can Tweety fly?" it is irrelevant if the agents disagree about matters which have no bearing on this question. Allowing certain inconsistencies can keep the information exchange between agents to a minimum with respect to the current task.

For maintaining proof consistent states in multi agent scenarios, we extend the BRTMS to distributed scenarios. We believe that using a JTMS as the basis for the DTMS is more advantageous than using an ATMS. The domain of an agent, that is its assumptions, premises and rules, can be expected to increase greatly in a multi agent cooperation process. An ATMS has to compute *all* newly arisen contexts. Fast query response time is overshadowed by the exponential cost of the ATMS labeling algorithm when maintaining a large domain. Instead, we extend the BRTMS of Chapter 2 to a Distributed Truth Maintenance System (DTMS). In contrast to classical, propositional logic based JTMSs, the first-order representation of beliefs and justifications in the BRTMS allows more expressive interaction between agents. We discuss the DTMS in a simple multi agent framework in order to demonstrate its features.

3.1 Agent

3.1.1 An Abstract Model

As the basis for designing real machine agents, we use the abstract agent model presented in [SMH90]. This model decomposes an agent into three main parts (Figure 3.1): The *agent*

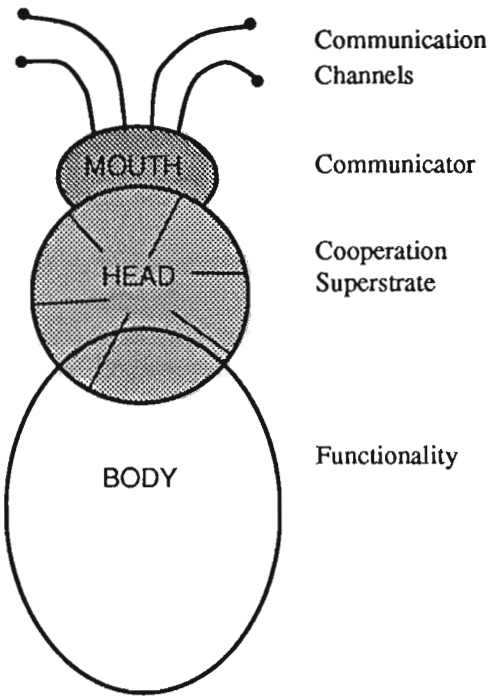


Figure 3.1: Parts of an Agent [SMH90].

mouth realizes the communication functionality of an agent. Via communication channels, it receives and passes messages to the agents head, and in the other direction, it is able to post agent's messages coming from its head. The agent's mouth has to be provided with sufficient network knowledge, such as physical addresses of agents or knowledge about how to get these addresses. Furthermore, a sophisticated design of an agent's mouth would be able to deal with a variety of communication formats (natural language, graphical representations, bitstreams ...) characterized by several different attributes. The authors of [SMH90] distinguish, for instance, the priority of a message, its type and the type of answer expected.

The *agent's head* incorporates mechanisms for cooperation and inference control. Thus, the head of an agent contains both meta knowledge of its own capabilities (autoepistemic knowledge), as well as meta knowledge of capabilities, status and behavior of other agents (epistemic knowledge). Designing the agents' head is a complex task. The following items are some additional features an agent's head should be provided with:

- knowledge about the state of the current task
- task decomposition algorithms
- facilities for inter-agent communication
- methods to change its cooperation behavior depending on globally available cooperation structures

We might regard the agent's head as the "mediator between the agent's individual functionality and the overall problem solving context."

Finally, the *agent body* realizes the basic problem solving functionality of an agent. The complexity of a body's functionality is not constrained: A sensor as well as whole expert systems or humans can be subsumed under the notion of 'agent body'.

For designing real machine agents, this concept has to be refined, in particular a precise definition of the mouth-head and head-body interfaces is necessary. As a first step, we present in the following the realization of one of the most important modules of the head, the *Meta Logic Unit (MLU)*, and its connection to the agent's body. In Figure 3.2 we see

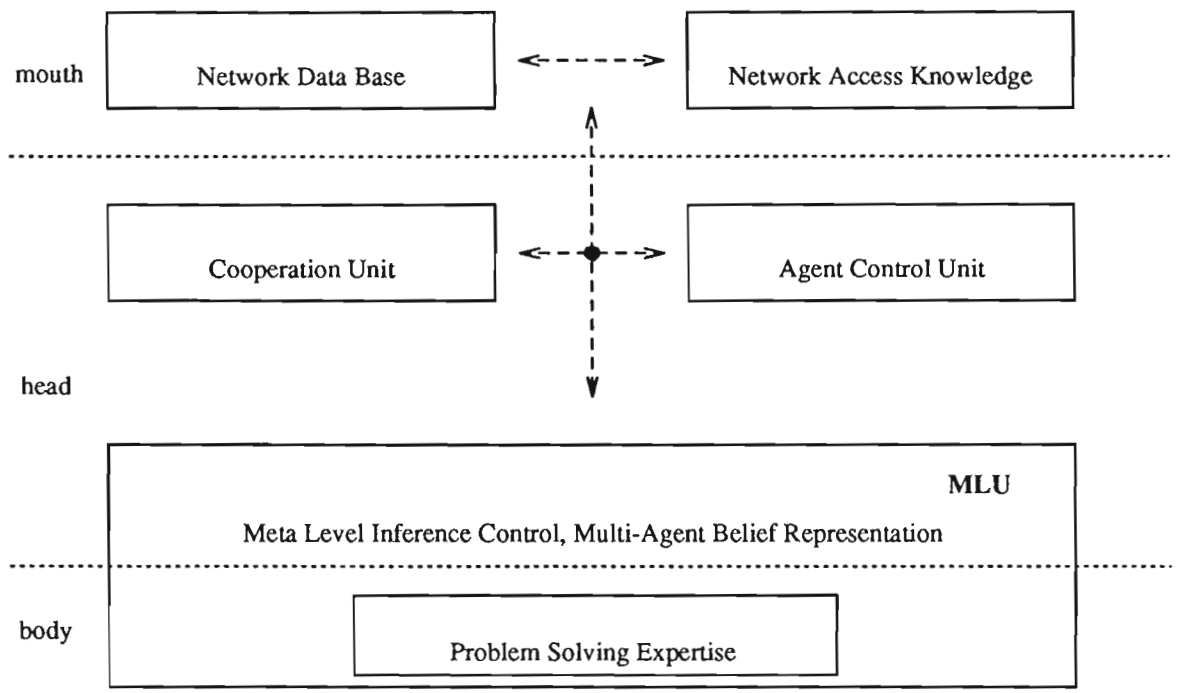


Figure 3.2: Agent Architecture.

the agent's head split into several units. Its main parts are the Cooperative Unit, the Meta Logic Unit and the Agent Control Unit.

3.1.2 Meta Logic Unit

The Meta Logic Unit is one of the central modules located in the agent's head. It controls and describes the inference mechanisms of the body, represents its own knowledge and that of other agents, allows reasoning with other agents' knowledge, provides a basis for all cooperation actions and much more. The following items summarize the general requirements:

- description of the agent's body functionality: In general, the functionality, as represented by the body, can be very complex. To establish an efficient interaction with other modules of the head, we need a description of the body's interface functions rather than all function definitions of the body itself. In addition, it might be useful for cooperation to know how fast or how reliably tasks might be processed by the agent's body.

  The meta level description of an agent's body depends greatly on its realization: Bodies realized by sensor systems might be described by their physical behavior, logic programs with the aid of meta logic programs, and databases by their top level functions. The difficulties of finding meta descriptions arise if a body is able to modify its behavior on his own. For example, a learning inference engine or a neural net might process tasks faster and more accurately over time. In such systems, a static meta description of the body is useless; the meta description has to continually reflect the current state of the body.

- enable reasoning with other agents' knowledge: The agent's body is designed to reason autonomously. In fact, we *want* the body to be independent from the head in order to build agents with preexisting hard- and software. On the other hand, such bodies are not able to reason with knowledge acquired from other agents. The MLU combines the functionality of the agent's body with functionality acquired by other agents and allows reasoning with this combined knowledge.

- supporting other modules: Because the MLU represents the entire knowledge and beliefs of an agent, it provides the basis for further knowledge processing, in particular it supports multi agent cooperation. The MLU should provide fast access to beliefs and inferences, expressive explanation of inferences and fast context switching. Furthermore, it is possible to design special agent bodies that take advantage of the explicit inference representation in the MLU.

Thus, designing the MLU of a machine agent yields a variety of issues. Some of them we want to discuss in more detail:

- what knowledge of other agents should be represented?
- how to represent knowledge of other agents
- how to deal with contradictory beliefs in different agents

3.1.3 Constructing a MLU with a DTMS

Our approach for providing a backward reasoning agent with a MLU is to extend the BRTMS of Chapter 2 to a Distributed Truth Maintenance System (DTMS). In general, basic facilities of JTMSs are also useful for MLUs. For instance, JTMSs generate a consistent state of beliefs, create explicit explanation trees for inferences and allow fast access to data inferred by the problem solver. On the other hand, JTMSs developed so far are inappropriate for representing inferences of other agents and their interface functions are insufficient for supporting multi agent cooperation. Furthermore, we have to redefine the notion of data consistency: Former TMSs establish global data consistency in a single agent scenario. In fact, we want to avoid global data consistency in a multi agent scenario. This would constrain the concept of autonomous agents: We could merge all beliefs of all agents together. Instead we introduce the concept of *proof consistency*. We categorize beliefs into those that might be inconsistent across some agents and those which might be held consistent across all participating agents. Actually, after a multi agent reasoning process is complete, the principal of proof consistency guarantees a consistent state of all those beliefs that have been involved in the reasoning process.

To provide an agent with a MLU we extend the BRTMS of Chapter 2. We give a framework for designing machine agents based on a backward reasoning system. We focus on the design of the agent's Meta Logic Unit and its body and present simple realizations of the Cooperative- and Agent Control Unit and mouth. Thus, we can establish some simple distributed scenarios that demonstrate the main features of our MLU.

3.2 DTMS

3.2.1 Beliefs in a DTMS

In the following, we consider a set of agents, each identified by an unique agent identifier. Each agent contains static predicates, local justifications and its own set of beliefs as discussed in Chapter 2. Additionally, an agent can *acquire* beliefs from other agents, or it can *transmit* beliefs to other agents. The next definitions make these notions more precise.

Definition 10 (Agent) *Let $\mathcal{P}$ be a set of program clauses, $\mathcal{B}$ be a set of beliefs and $\Psi = (\lambda, \mu, \nu, \xi)$ the state of $\mathcal{B}$. Then we call the triple $\alpha = (\mathcal{P}, \mathcal{B}, \Psi)$ an* agent. *The agent identifier α is logically equivalent to the symbol* true.

Definition 11 (Agent Rule) *Let $\mathcal{A} = \{\alpha_1 = (\mathcal{P}_1, \mathcal{B}_1, \Psi_1), \cdots, \alpha_n = (\mathcal{P}_n, \mathcal{B}_n, \Psi_n)\}$ be a set of agents. A* positive agent rule *of $\mathcal{P}_j$ is a justification of the form*

$$a \leftarrow \alpha_i$$

and a negative agent rule *is of the form*

$$a \leftarrow \neg\alpha_i$$

where a *denotes an ordinary atomic formula and* $i \in \{1, \cdots, n\}, (i \neq j)$.

Note, Definitions 10 and 11 imply that the positive agent rule $a \leftarrow \alpha_i$ is valid, if $\lambda(a) = \texttt{in}$ and the negative rule $a \leftarrow \neg\alpha_i$ is valid, if $\lambda(a) = \texttt{out}$ (with respect to the current state of beliefs of α_j).

This is what an agent rule is intended to do: A belief a that is inferred by an agent with a positive agent rule might be interpreted as "I believe in a, because agent α_i told me so" and a negative one as "I do not believe in a because neither I nor agent α_i can prove a"[1]. In these terms, a agent rule *represents* an inference in another agent.

Thus, the beliefs of an agent may categorized according to the following definitions.

Definition 12 (Beliefs in a Multi Agent System) *Let* $\mathcal{A} = \{\alpha_1 = (\mathcal{P}_1, \mathcal{B}_1, \Psi_1), \cdots, \alpha_n = (\mathcal{P}_n, \mathcal{B}_n, \Psi_n)\}$ *be a set of agents. We say a belief* $b_i \in \mathcal{B}_i$, *denoting the atom* l_i, *is*

(i) private *to* α_i, *if there is no belief* b_j *in* $\mathcal{B}_j$, *such that* l_i *can be unified with* l_j, $(i \neq j)$.

(ii) common *to* α_i *and* α_j, *if there is a belief* b_j *in* $\mathcal{B}_j$, *such that* l_i *can be unified with* l_j, $(i \neq j)$. *The status of* b_i *might be different from the status of* b_j.

(iii) transmitted *to agent* α_j, *if* $\mathcal{P}_j$ *contains either a positive agent rule of the form* $l_i \leftarrow \alpha_i$ *or a negative agent rule of the form* $l_i \leftarrow \neg\alpha_i$ $(i \neq j)$.

(iv) acquired *from agent* α_j, *if* $\mathcal{P}_i$ *contains either a positive agent rule of the form* $l_i \leftarrow \alpha_j$ *or a negative agent rule of the form* $l_i \leftarrow \neg\alpha_j$, $(i \neq j)$.

(v) mutual *to* α_i *and* α_j, *if* b_i *is transmitted to* α_j.

Transmitting a belief means passing an agent rule. Thus, an agent acquiring the agent rule is able to do its own, local, inferences based on this rule. In particular, the acquiring of a positive agent rule allows an agent to create a belief with the the same atom and status as in the transmitting agent. This is why we speak of transmitting beliefs rather than justifications.

Figure 3.3 shows an example for beliefs in a multi agent scenario in which belief Q is private

[1] Assuming, of course, the agent does not have its own valid justification for a.

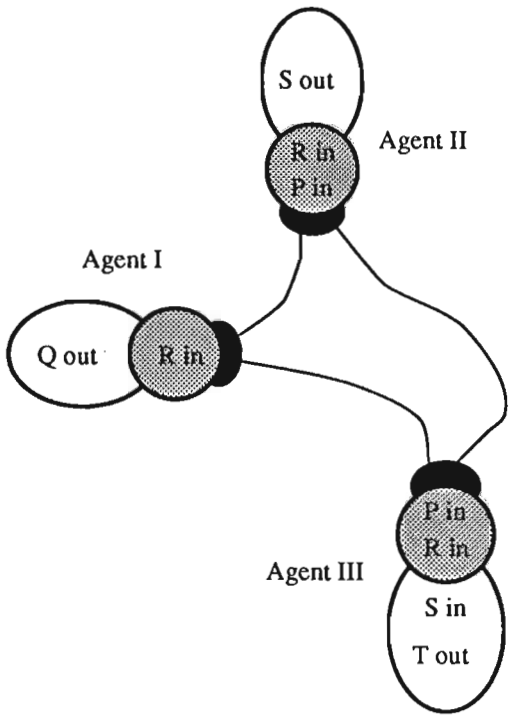


Figure 3.3: Common and Mutual Beliefs.

to Agent I, belief S is common to Agents II and III and belief R is mutual to I, II and III.

Many questions arise, when an agent can transmit its beliefs. We discuss them in the next section by defining terms for consistency in multi agent scenarios.

3.2.2 Consistency

Definition 11, in connection with Definition 8 of Section 2.2.2, allows us to define an agent's consistency. Informally, an agent is locally consistent if its own and acquired beliefs are consistent in accordance to its own set of program clauses.

Definition 13 (Local Consistency) *Let* $\{\alpha_1, \cdots, \alpha_n\}$ *be a set of agents. The agent* $\alpha_i = (\mathcal{P}, \mathcal{B}, \Psi)$ *is* locally consistent, *if its state* Ψ *is consistent.*

Definition 14 (Proof Consistency) *Let*
$\mathcal{A} = \{\alpha_1 = (\mathcal{P}_1, \mathcal{B}_1, \Psi_1), \cdots, \alpha_n = (\mathcal{P}_n, \mathcal{B}_n, \Psi_n)\}$ *be a set of agents.* $\mathcal{A}$ *is* proof consistent, *if the following conditions hold:*

(i) Each agent $\alpha_i \in \mathcal{A}$ is locally consistent.

(ii) If a belief $b \in \mathcal{B}_i$ is transmitted to agent $\alpha_j (i \neq j)$, then each ancestor $f_1, \cdots, f_m$ of b is either transmitted to agent α_j or acquired. b *and the acquired counterpart in agent α_j are either both* in *or both* out.

(iii) a belief $b_i \in \mathcal{B}_i$ that is acquired is not transmitted.

(iv) there is no set of beliefs $(b_0, \cdots, b_n) \in \mathcal{B}_1 \cup \cdots \cup \mathcal{B}_n$, such that $b_0 = b_n$ and for $i = 1, \cdots, n$, either

(a) $\lambda(b_i) =$ in *and b_{i-1} is in $\mu(b_i)$*
or

(b) there is a k *such that $b_{i-1} \in \mathcal{B}_k$ and $\alpha_k \in \mu(b_i)$.*

Definition 14 implies that we allow common beliefs in a multi agent system to be labeled differently. Condition (iv) guarantees a well founded set of mutual beliefs.

Example: Let $\mathcal{A} = \{\alpha_1 = (\mathcal{P}_1, \mathcal{B}_1, \Psi_1), \alpha_2 = (\mathcal{P}_2, \mathcal{B}_2, \Psi_2)\}$ be two agents with

$\mathcal{P}_1 = \{b \leftarrow a, a \leftarrow \alpha_2\}$
$\mathcal{B}_1 = \{b, a\}$ with $\lambda(b) = \lambda(a) =$ in, $\nu(b) = a$, $\xi(b) = \{\}$ and $\nu(a) = \alpha_2$, $\xi(a) = b$.

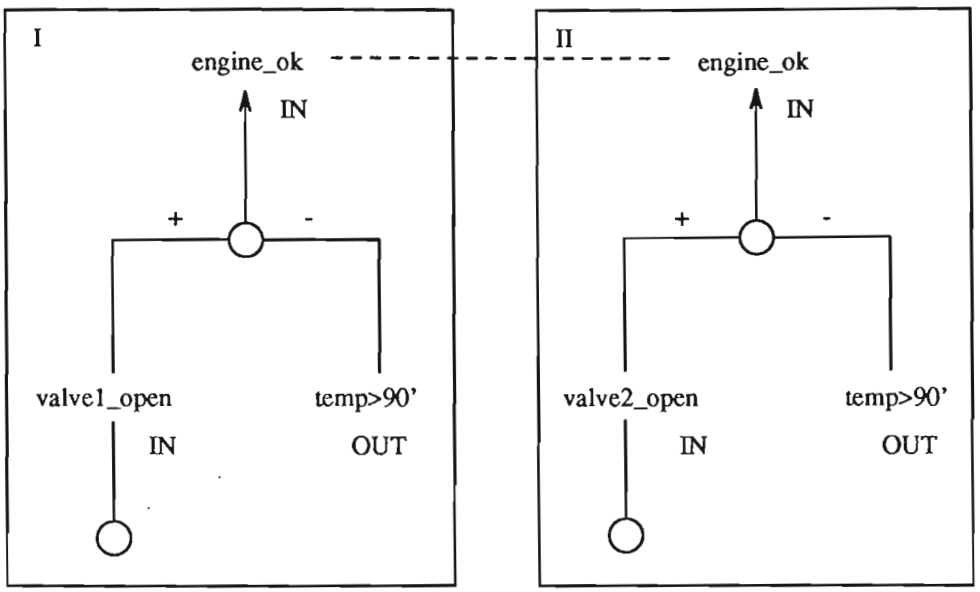
$\mathcal{P}_2 = \{a \leftarrow b, b \leftarrow \alpha_1\}$
$\mathcal{B}_2 = \{b, a\}$ with $\lambda(b) = \lambda(a) =$ in, $\nu(a) = b$, $\xi(a) = \{\}$ and $\nu(b) = \alpha_1$, $\xi(b) = a$.

We assume that belief b is transmitted from α_1 to α_2 and a from α_2 to α_1. $\{\alpha_1, \alpha_2\}$ is *not* proof consistent, because the well-foundedness condition is violated. But each single agent is locally consistent (see also Footnote 3 in Section 3.2.3).

Note that a belief is only transmitted once in the net. That is, an agent cannot transmit a belief that it has already acquired. On the other hand, the number of agents that can acquire another agent's belief is not constrained.

3.2.3 Example

To motivate and explain our definition of consistency in a multi agent scenario, we give an example of a typical situation. Consider Figure 3.4.



dynamic clauses of agent I:

```
dtms_node(engine_ok, in, (valve1_open , not 'temp>90'),
        [valve1_open, 'temp>90'], [], j1).
dtms_node(valve1_open, in, true, [true], [engine_ok], j2).
dtms_node('temp>90', out, true, [true], [engine_ok], _g55).

dtms_rule(engine_ok :- (valve1_open , not 'temp>90'), j1).
dtms_rule(valve1_open :- true, j2).
```

dynamic clauses of agent II:

```
dtms_node(engine_ok, in, (valve2_open , not 'temp>90'),
        [valve2_open, 'temp>90'], [], j1).
dtms_node(valve2_open, in, true, [true], [engine_ok], j2).
dtms_node('temp>90', out, true, [true], [engine_ok], _g55).

dtms_rule(engine_ok :- (valve2_open , not 'temp>90'), j1).
dtms_rule(valve2_open :- true, j2).
```

Figure 3.4: Simple Multi Agent Scenario.

Agent I and agent II are two autonomous control systems controlling the correct behavior of an engine. They reason that the engine works correctly, if either valve 1 or valve 2 is open but the temperature of the engine does not exceed $90°C$. Because the temperature is so important, we decide to use two autonomous cooperative agents, each able to detect a high temperature on its own. Furthermore, they each control a different valve. We want the agents to reason cooperatively about the belief `engine_ok` (represented by the dashed line in Figure 3.4), that is, we don't want the agents to have different statuses of `engine_ok`. Initially, querying the agents about `engine_ok` would yield the solution `engine_ok` is `in`, in accordance with both agents. Now, suppose agent II acquires a new justification for the belief `temp>90` because its sensor detects a temperature of more than $90°C$. This would cause agent II to relabel the beliefs `temp>90` and `engine_ok` `in` and `out`, respectively, in contradiction to the corresponding labels of agent I.

The only known distributed labeling algorithm [BH90], would now create the symbol `in`[2] for belief `engine_ok` of agent II, because agent I still has a valid justification for `engine_ok`. (Agent I's temperature sensor did not recognize the high temperature for some reason.) However, we do not want the system to believe that the engine is still functioning. The reason for this undesired behaviour is: In the scenario, only the *result* of an inference is shared between agents, but not *reasons* for it. Agent II does not 'know' that agent I continues to believe the engine is ok because it has no reason to believe the temperature is higher than $90°C$. Thus, agent II does not inform agent I about his recognition of the high temperature and agent I will dominate, even though it has less information.

In our terms, the created state above would *not* be proof consistent, because condition (ii) of Definition 14 is violated. We require the transmission of all ancestors of a transmitted belief as well. If we want the agents to yield a cooperative solution about a belief b in our system, the following will happen. Via a cooperation process, the agents will select one single proof for b of one single agent. We say, this agent is *responsible* for b. In this agent, *all* ancestors $f_1 \cdots f_n$ of b will be marked as transmitted. Furthermore, all of the agents involved in the cooperation process will acquire agent rules represented as

`dtms_rule ((`f_i `:- responsible_agent),` id_i`).`

or

`dtms_rule ((`f_i `:- not responsible_agent),` id_i`).`

for each ancestor $f_1 \cdots f_n$. The positive agent rule is added if the corresponding ancestor in the responsible agent is `in`, otherwise the negative one is added.[3]

---

[2] Precisely, the labeling algorithm as described in [BH90] will create the symbol `external`. It is logically equivalent to the symbol `in`, but denotes that the valid justification for the belief is in another agent.

[3] Thus, we could represent the example of Definition 14 in Prolog code as follows:

agent I:

```
dtms_rule((b:-a), j1).
```

The following clauses show the effect of an agent query in our engine example (the algorithms will be discussed in more detail in 3.3.5 and 3.3.7):

dynamic clauses of agent I:

```
transmitted_node(engine_ok, [agent_2]).
transmitted_node(valve1_open, [agent_2]).
transmitted_node('temp>90', [agent_2]).

dtms_node(engine_ok, in, (valve1_open , not 'temp>90'),
        [valve1_open, 'temp>90'], [], j1).
dtms_node(valve1_open, in, true, [true], [engine_ok], j2).
dtms_node('temp>90', out, true, [true], [engine_ok], _g55).

dtms_rule(engine_ok :- (valve1_open , not 'temp>90'), j1).
dtms_rule(valve1_open :- true, j2).
```

dynamic clauses of agent II:

```
dtms_node(valve2_open, in, true, [true], [], j2).
dtms_node(engine_ok, in, agent_1, [agent_1], [], j3).
dtms_node(valve1_open, in, agent_1, [agent_1], [], j4).
dtms_node('temp>90', out, agent_1, [agent_1], [], _g55).

dtms_rule('temp>90' :- not agent_1, j5).
dtms_rule(valve1_open :- agent_1, j4).
dtms_rule(engine_ok :- agent_1, j3).
dtms_rule(engine_ok :- (valve2_open , not 'temp>90'), j1).
dtms_rule(valve2_open :- true, j2).
```

We assume agent I is responsible for the belief `engine_ok`. Agent I transmitted its beliefs `engine_ok, valve1_open, temp>90` to agent II. For each transmitted belief, there is

---

```
dtms_rule((a:-agentII), j2).

dtms_node(b, in, a, [a], [], j1).
dtms_node(a, in, agentII, [agentII], [b], j2).
```

agent II:

```
dtms_rule((a:-b), r1).
dtms_rule((b:-agentI), r2).

dtms_node(a, in, b, [b], [], r1).
dtms_node(b, in, agentI, [agentI], [a], r2).
```

an entry of the form `transmitted_node (node, agent_list)` representing, in its second argument, all agents to which the belief has been transmitted. The three DTMS-nodes represents its current set of beliefs, just as described in Chapter 2.

Agent II, however, acquired three agent rules from agent I. These are the rules with rule ids $j1 \cdots jn$. We see from the current set of beliefs of agent II that the belief `engine_ok` is now supported by agent I and not by its own justification j1. Nevertheless, j1 of agent II is still valid.

The agents shown above are *proof consistent*: Each agent is locally consistent, all mutual beliefs have the same status and all ancestors of transmitted beliefs are also transmitted. Furthermore, the set of in beliefs is well founded w.r.t. condition (iv) of Definition 14.

Now suppose agent II acquires a new valid justification for the belief `temp>90` which disrupts proof consistency. Because `temp>90` changes its status originally supported by agent I, agent II 'tells', that is transmits, its new belief to agent I via passing the agent rule `temp>90 :- agentII`. Acquiring a new valid justification from agent II for its belief `temp>90`, agent I relabels downstream its belief `engine_ok` to status `out`. This involves belief `engine_ok` in agent II again, because `engine_ok` had been transmitted from agent I to agent II. In these terms, we get the labeling of `engine_ok` to `out` in both agents:

dynamic clauses of agent I:

```
transmitted_node(engine_ok, [agent_2]).
transmitted_node(valve1_open, [agent_2]).

dtms_node(valve1_open, in, true, [true], [], j2).
dtms_node('temp>90', in, agent_2, [agent_2], [engine_ok], j3).
dtms_node(engine_ok, out, not valve1_open ; 'temp>90',
        ['temp>90'], [], _g55).

dtms_rule('temp>90' :- agent_2, j3).
dtms_rule(engine_ok :- (valve1_open , not 'temp>90'), j1).
dtms_rule(valve1_open :- true, j2).
```

dynamic clauses of agent II:

```
transmitted_node('temp>90', [agent_1]).

dtms_node(valve2_open, in, true, [true], [], j2).
dtms_node(valve1_open, in, agent_1, [agent_1], [], j3).
dtms_node('temp>90', in, true, [true], [engine_ok], j6).
dtms_node(engine_ok, out, ((not valve2_open ; 'temp>90') , agent_1),
        ['temp>90', agent_1], [], _g55).

dtms_rule(engine_ok :- not agent_1, j5).
```

```
dtms_rule('temp>90' :- not agent_1, j4).
dtms_rule(valve1_open :- agent_1, j3).
dtms_rule(engine_ok :- (valve2_open , not 'temp>90'), j1).
dtms_rule(valve2_open :- true, j2).
dtms_rule('temp>90' :- true, j6).
```

As we see in this example, *both* agents are responsible for the status of `engine_ok` being out: Agent I inferred `valve1_open` and transmitted it to agent II, while vice versa agent II inferred `temp>90` and transmitted it to agent I.

So far we have discussed basic terms in a multi agent scenario. Our concept allows an agent to reason about its own justifications and rules acquired by other agents while guaranteeing a defined degree of consistency. This degree of consistency ensures solutions to queries that are correct and consistent with all agents involved in the reasoning process with a minimal exchange of knowledge. Thus, our definition of *proof consistency* can be seen as a trade-off between exchanging as little knowledge as possible and guaranteeing consistent solutions with respect to all involved agents.

Furthermore, an agent does not have to know about the structure and the dependencies of another agent's proof; only the ancestors and the statuses of the ancestors has to be transmitted. This allows an agent to store and access knowledge of other agents in a very efficient way.

3.3 DTMS in a Multi Agent Scenario

3.3.1 DTMS Architecture

Figure 3.5 shows the architecture of the DTMS. The extension allowing for distributed truth maintenance does not alter the basic architecture of the BRTMS. However, the functionality of the meta level is greatly enhanced. We have to add predicates for incorporating beliefs of other agents, for transmitting beliefs to other agents and for establishing proof consistency in the whole scenario. Furthermore, a complete algorithm has to detect circularities in the set of beliefs spanning several agents in order that execution will terminate.

Actually, there are no predicates for synchronizing the exchange of beliefs in the scenario in the DTMS; this is part of the cooperation process. The framework here is appropriate for designing scenarios with logically distributed agents. But it is necessary that only one agent be active at a time in order to prevent deadlocks. In fact, there are tasks that could be done in parallel without changing the algorithm. We will mentioned this later when discussing the algorithm.

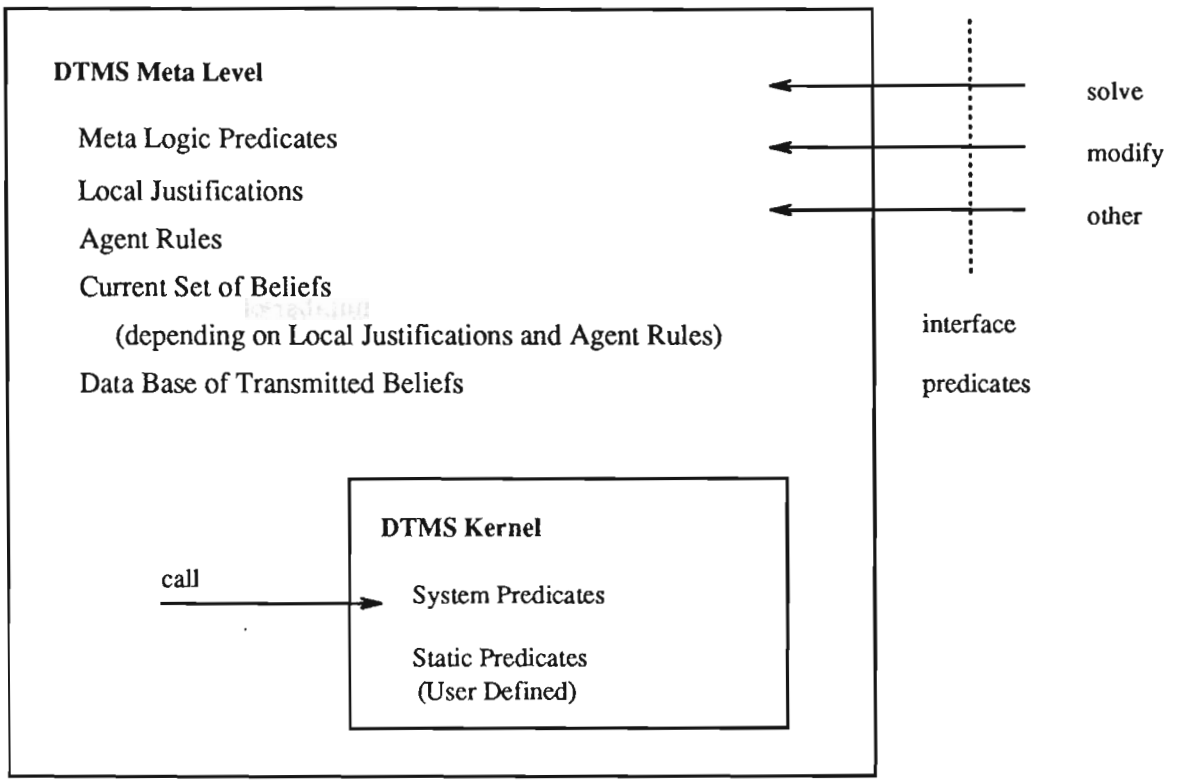


Figure 3.5: DTMS Architecture.

3.3.2 Level of Consensus

In Section 2.2.1 we discussed the basic features of the kernel. In multi agent scenarios, we can use the DTMS kernel advantageously in order to increase the efficiency of belief exchange among agents. Definition 14 requires exchanging beliefs along with its ancestors. But, in general, the inference chain represented in a belief might be long. Actually, there is no point for transmitting conclusions as part of its ancestors which will have the same status in all agents at all times. In order to define a level, on which no cooperation is necessary, we can use the kernel. Predicates defined here are evaluated without performing truth maintenance, that is, the result of a kernel call will be represented in a belief *without* representing the reasons for it. In these terms, the exchange of beliefs can be greatly minimized without violating the principal of proof consistency.

Note that the design of the DTMS kernel has to be done very carefully. Unpredictable behaviour may occur if two agents infer different statuses of the same query with kernel predicates.

As mentioned before, we will discuss the DTMS embedded in a simple agent framework

containing some essential properties of an agent. In the following, we will discuss a scenario of such agents. We start the discussion by setting up the scenario. Step by step, we fulfill the abstract remarks of the previous sections with the counterparts of a backward reasoning agent.

3.3.3 Multi Agent Scenario

A multi agent scenario can be set up with an unlimited number of agents. Here, each agent has it own id and can access the ids of other agents from a network database. For instance, if set a scenario is created with the agents {bond, hari, smiley}, agent bond will be provided with the following three facts:

```
whoami(bond).

agent(hari).
agent(smiley).
```

An agent (or the user) can access another agent via a call of the following form:

```
call (Goal, Agent_id)
```

This call will resolve `Goal` with the set of program clauses denoted by `Agent_id` in the usual manner. The set of predicates that are visible outside an agent will be defined later. In addition, the setup defines all valid agent identifiers allowing the DTMS to distinguish ordinary symbols from those describing another agent. This realizes the basic functionality of an agent's mouth.

This has setup a scenario with several agents, each with its own network knowledge and its own problem solving expertise. Apart from that, the agents are equal to each other.

Furthermore, we require the initial state of the scenario to be proof consistent.

3.3.4 Top Level Predicates

Each agent of the net can be queried in two different ways. A *local* query concerns only a single agent. That is, the agent resolves the query with its own set of program clauses (possibly containing acquired agent rules) but without interacting with other agents. Actually, a local query is just a call to a modified `dtms_solve/5`[4] as described in Chapter 2.

[4] See Figure 3.7

The second way an agent can be queried is by calling the predicate agent_query/3. In general, this invokes several agents in the reasoning process. Figure 3.6 shows its definition. agent_query/3 is called with an atomic formula as its first argument. The arguments

```
agent_query(Atom, Status, Support) :-
     knowledge(Atom, Agents),
     discuss(Atom, Agents, Supportproof, Supportagent, Otheragents),
     call(transmit_proof(Supportproof, Otheragents), Supportagent),
     install_proof_globally(Supportproof, Otheragents, Supportagent),
     call(dtms_solve (Atom, Status, Support, db, no), Supportagent).
```

Figure 3.6: Agent Query.

Status and Support will be bound to the status and support, respectively, of the belief, representing Atom, that the DTMS will create.[5] First, an agent finds all[6] agents of the net whose set of predicates contain a definition for Atom. This is done in the second argument of the predicate knowledge/2. No other agent will take part in the following reasoning process; this may even include the queried agent itself. In this case, the queried agent is called a *moderator* of the reasoning process between other agents. If Atom is not defined anywhere knowledge/2 will fail.

The predicate discuss/5 defines the interface to the cooperation unit in the agent's head. A call to discuss/5 returns in the fourth argument, Support_agent, the id of the responsible agent for the belief representing Atom and in its third argument, Support_proof, all ancestors of Atom with the corresponding states. The fifth argument, Otheragents, returns all other agents that have been involved in the cooperation process. These are precisely those agents that have to acquire the proof of Supportagent.

A discussion in our scenario is implemented very simply: The strategy for selecting a responsible agent for Atom is to search for the agent with the most ancestors with status in of the belief representing Atom. Certainly, this strategy is too simple for use in practise, but it is sufficient for demonstrating the DTMS features. In particular, the DTMS supports the cooperation unit in generating merged proofs for a belief incorporating rules and beliefs of different agents, as in the previous example. However, the cooperation unit is constrained to select one agent to be responsible for Atom.

The next call in agent_query/3 marks, in the responsible agent, all ancestors of the belief denoting Atom as transmitted. The definition of transmit_proof/2 is straightforward; it simply asserts or modifies facts of the form transmitted_node(Atom, Agentlist) for each

[5] A more sophisticated definition of agent_query/3 could allow complex goals in queries. This requires implementing a *task decomposition facility* that generates subgoals for specific agents.

[6] Certainly, it might be useful to involve only some agents in the reasoning process instead of all which have knowledge about the current goal. knowledge/2 is, like the following discuss/5, user-defined and can be modified in a such a way. In particular, knowledge/2 could work in parallel.

atom of Supportproof.

install_proof_globally/3 allows Otheragents to acquire the members of Support_proof transmitted by Support_agent. But, in general, the elements of Support_proof conflict with the belief statuses of the ancestors for the belief representing Atom in one of Otheragents. That is, it is possible that a status of a transmitted belief is in, but the local counterpart in an acquiring agent is out, or vice versa. Thus, additional work needs to be performed when transmitting a belief to another agent as described in the algorithm of section 3.3.5.

At the end of agent_query/3, we query the responsible agent for getting the status and support of Atom via dtms_solve/5. Actually, the call to dtms_solve/5 at this point is reduced to simply accessing the stored beliefs, because the previous reasoning process before ensures the creation of a corresponding belief for Atom in all involved agents.

In our agent model, the definition of agent_query/3 is located in the head's Control Unit and discuss/5 in the Cooperation Unit of the agent's head (see Figure 3.2). discuss/5 is the only predicate located in the Cooperation Unit, thus, it simultaneously defines the interface between the Cooperation and the Control Unit.

The interface between the MLU and the Control Unit is defined via the three predicates transmit_proof/2, install_proof_globally/3 and dtms_solve/5. Note that the latter meta predicate defines a set of interface predicates, including all predicates concerning the agent's problem solving expertise. We see from the location of the predicate install_proof_globally/3 in the DTMS, that an agent's DTMS *directly* accesses the DTMS of other agents in order to establish proof consistency in the agent scenario.

3.3.5 Algorithm for Transmitting a Belief

The following presents the algorithm for transmitting a belief as announced above.

pre: proof consistent state of α and $\hat{\alpha}$, such that the set of beliefs in α contain the belief b_α representing the atom l. Its set of justifications does not contain circular dependencies.[7]

action: $\hat{\alpha}$ acquires belief b_α from α.

post: proof consistent state of α and $\hat{\alpha}$. $\hat{\alpha}$ acquired the belief b_α from α. This implies the beliefs b_α and $b_{\hat{\alpha}}$ are mutual to $\{\alpha, \hat{\alpha}\}$.

(i) status (b_α) = in, but there is a common belief $b_{\hat{\alpha}}$ in $\hat{\alpha}$, with status out.

In this case, $\hat{\alpha}$ does not have its own valid justification for l. Thus, asserting the (valid) agent rule (l :- α) via the predicate assert_justification/2 in $\hat{\alpha}$, will change the

[7] For a discussion of circular dependencies, see Section 3.4.

status of $b_{\hat{\alpha}}$ from out to in. If there are consequences of $b_{\hat{\alpha}}$, $\hat{\alpha}$ invokes downstream propagation of the repercussions of $b_{\hat{\alpha}}$, probably by invoking the algorithm in 3.3.7.

(ii) status (b_α) = out and there is a belief $b_{\hat{\alpha}}$ in $\hat{\alpha}$ common to α with status in.

It is *not* sufficient to assert the agent rule (`l :- not` α), because the local justification for l in $\hat{\alpha}$ is still valid and the relabeling procedure in $\hat{\alpha}$ will not alter $b_{\hat{\alpha}}$'s status to out. Instead, we have to retract all valid justifications for l in $\hat{\alpha}$ via `retract_justification/1` by successively relabeling its repercussions downstream. This might invoke the algorithm of 3.3.7. After that, we assert the (invalid) agent rule (`l :- not` α). The relabeling mechanism of $\hat{\alpha}$ will guarantee that the supporting list of $b_{\hat{\alpha}}$ will contain the agent identifier α.

(iii) status (b_α) = out and there is a belief $b_{\hat{\alpha}}$ in $\hat{\alpha}$ common to α with status out.

This equals (ii) without the preceding retraction of all valid justifications for l in $\hat{\alpha}$. Thus, invoking downstream relabeling is not necessary.

(iv) status (b_α) = in and there is a belief $b_{\hat{\alpha}}$ in $\hat{\alpha}$ common to α with status in.

Similar to (i), we assert the agent rule (`l :-` α) at the beginning of the agent's justification database. In these terms, agent rules will get the highest priority of all justifications, ensuring that these rules will support the beliefs in an agent, if possible. No downstream relabeling is necessary.

(v) b_α with status in is private to α.

We assert the positive agent rule (`l :-` α) in the set of justifications of $\hat{\alpha}$ via `assert_justification/2`. $\hat{\alpha}$ will create a belief $b_{\hat{\alpha}}$ with status in.

(vi) b_α with status out is private to α.

We assert the negative agent rule (`l :- not` α) in the set of justifications of $\hat{\alpha}$ via `assert_justification/2`. $\hat{\alpha}$ will create a belief $b_{\hat{\alpha}}$ with status out.

3.3.6 Meta Predicates

In order to compute an agent's state of beliefs that might depend on agent rules, the predicate `dtms_solve/5` of Chapter 2 must be extended. Following Definition 10, this is easily done: Because an agent identifier is logically equivalent to the symbol `true`, all beliefs matching the head of a positive agent rule can be labeled in with the support field `[agent_id]`[8]. Thus, modifying the definition of `dtms_solve/5` as shown in Figure 3.7 will allow correct resolution of goals with predicates including agent rules. Note that the evaluation of negative agent rules can be subsumed under the evaluation of negated goals in general (see clauses 3 and 4 in Figure 2.7).

[8]The DTMS distinguishes agent identifiers from ordinary symbols with the aid of the facts `agent(agent_id)` as introduced in the beginning of Section 3.3.3.

```
                                    ⋮
dtms_solve ((not Goal), in, Constraint, Mode, Rem) :- !,
       dtms_solve (Goal, out, Constraint, Mode, Rem).
       agent(Agent), !.
dtms_solve (Agent, _, Agent, Mode, Rem) :-
       agent(Agent), !, fail.
dtms_solve (Goal, in, Constraint, Mode, Rem) :-
       Mode = tms,
       clause (dtms_node (Goal, in, Constraint, _, _, _)).
                                    ⋮
```

Figure 3.7: Modified `dtms_solve/5`.

Additionally, we have to modify the relabeling mechanism of the BRTMS, because belief relabeling in the BRTMS has so far only been applicable to a single agent scenario. The following algorithm proceeds from a proof consistent state of agents, in that one agent changes the status of a belief that is either transmitted or acquired. The goal of this algorithm is to reestablish proof consistency.

3.3.7 Algorithm for Relabeling Mutual Beliefs

pre: proof consistent state of agents α and $\hat{\alpha}$ such that the set of beliefs in α contain a belief b_α representing the atom l. Its set of justifications does not contain circular dependencies[9].

action:

1. α changes a belief b_α that it has transmitted to $\hat{\alpha}$ by modifying its local justification set.

   or

2. $\hat{\alpha}$ changes a belief $b_{\hat{\alpha}}$ that it has acquired from α.

post: proof consistent state of α and $\hat{\alpha}$.

1. α changes a belief b_α that it has transmitted to $\hat{\alpha}$ by modifying its local justification set.

   (a) α changes the support of b_α, but the status of b_α remains the same.

[9] For a discussion of circular dependencies, see Section 3.4.

The arisen situation conflicts with condition (ii) of Definition 14, if the new support contains beliefs, not yet transmitted to $\hat{\alpha}$. In this case, we transmit the new ancestors for b_α to $\hat{\alpha}$ following the Algorithm 3.3.5. Actually, there is no reason that the old ancestors of b_α have to be marked as transmitted any longer. Without invalidating the reestablished proof consistency, we could retract all the corresponding entries in the responsible agent and the agent rules in $\hat{\alpha}$. But, in most cases, $\hat{\alpha}$ inferred further beliefs depending on these agent rules. Thus, retracting these agent rules would invoke a superfluous relabeling of beliefs in $\hat{\alpha}$.

(b) α changes the status of b_α from `in` to `out`.

This causes condition (ii) of Definition 14 to be violated in any case. Condition (i) is violated in α, if there are consequences of b_α.

In order to satisfy condition (ii) again, we will distinguish two cases: 1(b)i $\hat{\alpha}$ still has its own valid justification for b_α, and 1(b)ii $\hat{\alpha}$ does not have a valid justification for b_α.

i. in this case, the DTMS in α makes $\hat{\alpha}$ responsible for the belief. That is, the entry `transmitted_node (b, Agents_a)` in α will be deleted and the fact `transmitted_node (b, Agents_b)` asserted in $\hat{\alpha}$. The list `Agents_b` results from exchanging the id $\hat{\alpha}$ with α in the list `Agents_a`. Furthermore, $\hat{\alpha}$ transmits all its ancestors for $b_{\hat{\alpha}}$ to all members of `Agents_b` as described in Algorithm 3.3.5 and α will retract all corresponding (positive) agent rules in all members of `Agents_a`[10]. After that, b_α is labeled `in` again with support $[\hat{\alpha}]$. This implies that α will be locally consistent, because the status of b_α is the same as before. Furthermore, the transmission of the ancestors of $b_{\hat{\alpha}}$ to α causes condition (ii) to be satisfied.

ii. we retract the positive agent rule in $\hat{\alpha}$ and assert the negative one. Because $\hat{\alpha}$ lacks a valid justification in $\hat{\alpha}$, the relabeling Algorithm of $\hat{\alpha}$ will set the status of b to `out`. But this might also cause $\hat{\alpha}$ to be locally inconsistent, forcing a relabeling downstream of repercussions of $b_{\hat{\alpha}}$. When relabeling of $\hat{\alpha}$ is complete, α relabels downstream its repercussions of b_α and transmits the new ancestors of b_α to $\hat{\alpha}$, following Algorithm 3.3.5.

(c) α changes the status of b_α from `out` to `in`.

α is now the *only* agent that has a valid justification for b in the whole scenario. We retract in $\hat{\alpha}$ the negative agent rule and assert the positive counterpart. In general, this makes $\hat{\alpha}$ locally inconsistent. To reestablish proof consistency, we follow the procedure of 1(b)ii.

2. $\hat{\alpha}$ changes belief $b_{\hat{\alpha}}$ acquired from α.

(a) $\hat{\alpha}$ changes the status of $b_{\hat{\alpha}}$ from `out` to `in`. Changing an acquired belief from `out` to `in` equals 1(b)i above: The DTMS of $\hat{\alpha}$ will make $\hat{\alpha}$ responsible for b.

[10] In these terms, an `in` belief in an agent dominates an `out` belief, that denotes the same atom in another agent. Alternatively, we could make α responsible for the belief b. In this case, we have to retract all valid justifications in all agents to which the belief was transmitted. Both possibilities make sense in certain cases, so that a preceding cooperation process should decide, which alternative is to be used.

The entry `transmitted_node (b, Agents_a)` in α will be deleted and the fact `transmitted_node (b, Agents_b)` asserted in $\hat{\alpha}$. The list `Agents_b` results from exchanging the id $\hat{\alpha}$ with α in the list `Agents_a`. Furthermore, $\hat{\alpha}$ transmits all its ancestors for $b_{\hat{\alpha}}$ to all members of `Agents_b` as described in algorithm 3.3.5. After that, b_α is labeled `in` with support `[`$\hat{\alpha}$`]`. But, in contrast to 1(b)i, α will be locally inconsistent, because the status of b_α changes. Thus, a downstream propagation of the consequences of b_α will be performed, possibly by invoking relabeling of further mutual beliefs. Finally, $\hat{\alpha}$ will relabel downstream its repercussions of $b_{\hat{\alpha}}$[11].

(b) Other cases are actually not possible, due to the fact that agent rules are asserted in the beginning of an agent's justification database. Thus, the `dtms_solve/5` procedure of an agent will always justify acquired beliefs with a agent rule and not with its own, if possible. But, as mentioned in Chapter 2, the support field of a belief only changes, if its supporting justification changes. Besides, one single agent is certainly not allowed to retract acquired agent rules, this can be done only by the responsible agent.

3.3.8 Example

Consider again Figure 3.4 showing the set of beliefs and justifications in two agents after the set up procedure as described in 3.3.3 is complete[12]. If we want agent 1 to yield a *cooperative solution* about the goal `engine_ok`, we do the following query:

```
call(agent_query(engine_ok, St, Su), agent_1).
```

Thus, `agent_query/3` in agent 1 is invoked. After calling `knowledge/2`, `Agents` will be bound to the list `[agent_1, agent_2]`, because both agents have a definition for the goal `engine_ok`. The cooperation via `discuss/5` yields the following variable bindings.

```
discuss(engine_ok, [agent_1, agent_2],
        [engine_ok - in, valve1_open - in, 'temp>90' - out],
        agent_1, [agent_2])
```

[11] This might cause unnecessary relabeling, see 3.4

[12] The facts

```
whoami (agent_1).
agent (agent_2).
```

in agent 1 and

```
whoami (agent_2).
agent (agent_1).
```

in agent 2 are not shown.

Agent 1 is now *responsible* for `engine_ok`[13] and generates the `Supportproof` for it in the third argument. After calling `transmit_proof`, all ancestors (the members of the list `Supportproof`) are transmitted to agent 2 as done in the example of 3.2.3.

The following call to `install_proof_globally` invokes algorithm 3.3.5: Transmitting `engine_ok` from agent 1 to agent 2 invokes (iv) of 3.3.5; `valve1_open` is private to agent 1, thus (v) is invoked, while transmitting `'temp>90'` invokes (iii). Because there are no circular dependencies in the set of justifications, algorithm 3.3.5 will terminate and create the proof consistent state as shown in 3.2.3.

Finally, the call to `dtms_solve/5` unifies `St` and `Su` with the status and support of the belief `engine_ok` of agent 1.

Now suppose agent 2 justifies its node `'temp>90'` with a premise justification making the agents inconsistent. 2a of algorithm 3.3.7 will perform the following: The DTMS of agent 2 transmits the agent rule `'temp>90' :-` *agent_2* to agent 1. This implies the DTMS of agent 1 will make its belief representing `'temp>90'` `in` and causes relabeling of `engine_ok` to `out`. Because `engine_ok` is transmitted to agent 1, algorithm 3.3.7 is invoked a second time. 1(b)ii of 3.3.7 will make `engine_ok` of agent 2 `out`. Because there are no consequences of `engine_ok` in agents 1 and 2, we are done after agent 2 relabels downstream the consequences of its belief `'temp>90'`.

The resulting beliefs are shown in the example of 3.2.3.

3.4 Discussion

The algorithms of 3.3.5 and 3.3.7 describe how two interacting DTMSs reestablish proof consistency, if one agent disrupts a proof consistent state by modifying its justification database. Changes in the statuses of mutual beliefs cause its propagation to other relevant agents. Furthermore, the transmission of all ancestors of mutual beliefs according to Definition 14 is performed.

The algorithms of 3.3.5 and 3.3.7 will terminate, if the following conditions are satisfied:[14]

1. there are no circularities in one agent's local set of justifications.
2. there are no circularities in the set of justifications across several agents.

We discussed 1. in Section 2.3.1 already. The discussion of 2. is similar, because analogous

[13] Both agents have the same number of in-ancestors for `engine_ok`. Because agent 1 is queried first, it will be responsible.

[14] Note, these conditions imply the well-foundedness of beliefs w.r.t (iii) of Definition 8 and (iv) of Definition 14.

arguments hold. Figure 3.8 shows a proof consistent state of agents I and II. All beliefs are labeled out. It is easily seen that supporting b in agent I with a valid justification will not allow creating of a proof consistent state. But the current version of the DTMS will not terminate, because the *repercussions of an acquired belief â contain a belief that is in the ancestors of the corresponding transmitted belief* a. A complete algorithm would be coNP complete, see Section 2.3.1.

Avoiding circularities, we can apply our algorithm to scenarios with more than two agents: If a belief is transmitted to more than one agent, we apply the algorithm successively to all the relevant agents. Finally, the created state of beliefs will be proof consistent. Indeed, unnecessary computations of states may occur: Suppose we have the situation presented in Figure 3.9: Obviously, the status of *d* in agent I is `in`, independent of the status of *b* in agent I. But if *b* changes status from `out` to `in`, our algorithm will cause *c* in agent II to be labeled `out`. Because d now has no valid justification (e is still out), the expensive relabeling of the consequences of d in agent I is invoked. Once that is complete, agent I relabels downstream the consequences of its belief b and invokes the relabeling of d a second time. Modification of the algorithm in order to avoid such cases is a subject for further research.

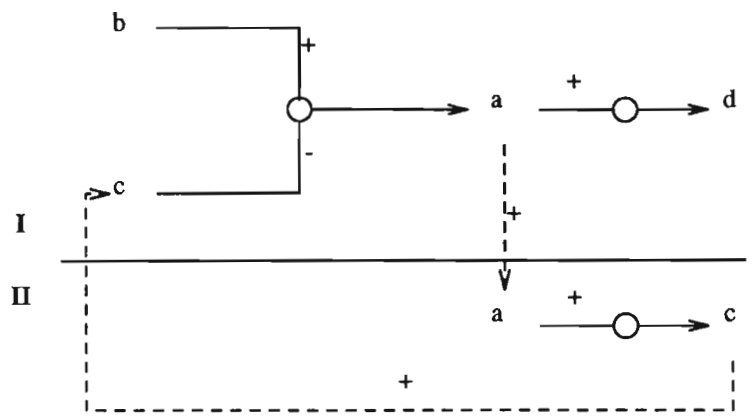


Figure 3.8: Odd Loop in a Multi Agent Scenario.

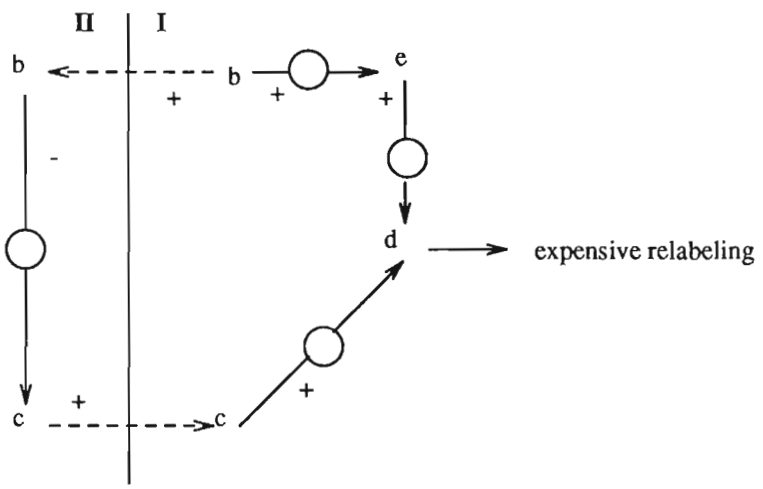


Figure 3.9: Example of Unnecessary Relabeling.

Chapter 4

Conclusion

Zwei mal zwei gleich vier ist Wahrheit.
Schade, daß sie leicht und leer ist,
Denn ich wollte lieber Klarheit
Über das, was voll und schwer ist.

Twice two equals four: 'tis true
But too empty, and too trite.
What I look for is a clue
To some matters not so light.[1]

— W. Busch, Schein und Sein

In Chapter 3 we presented a new concept supporting reasoning among autonomous interacting agents in distributed scenarios and demonstrated its usefulness in a concrete implementation. Central to this concept is the new term *proof consistency* which clearly defines a state of consistency of mutually dependent beliefs across different agents. This state is characterized by exchanging inferences *and* their foundations. We showed that in contrast to previous approaches, our definition of consistency allows agents to reason in a more complex way. Information, lost in former approaches, will now be propagated to all relevant agents: Because one agent knows the foundations of an acquired inference of another agent, it can inform this agent when a foundation becomes invalid. We showed the agents will not be overwhelmed with information. It is sufficient only to exchange a special, minimal representation of inferences between agents.

Our algorithm for establishing proof consistency in a multi agent scenario is based on a first-order truth maintenance system, *BRTMS*, introduced in Chapter 2. In contrast to propositional based TMSs, a first order representation of beliefs provides for more expressive interaction between agents while simultaneously guaranteeing a precise theoretical background. Furthermore, our approach of defining a TMS as a variant of a Prolog meta interpreter relieves the application designer from evaluation control tasks; this is delegated to meta evaluation by the BRTMS. In addition, meta logic control allows the application designer to distinguish between goals on which truth maintenance will be performed and

[1] This translation is due to Karl R. Popper in "Conjectures and Refutations: The Growth of Scientific Knowledge", Harper Torchbooks TB 1376. Popper said about his own translation that it renders "it perhaps more like a nursery rhyme than Busch intended".

goals which remain static. This yields higher performance and avoids unnecessary exchange of information between agent's.

Nevertheless, meta logic control of goal evaluation incurs additional costs that can overshadow the performance of the BRTMS in some cases. For single agent contexts, the BRTMS is used to best advantage on applications where certain goals can be expected to be selected frequently. We are still confronted with the *lemma generation problem* (see Section 2.3.2) that is not solved in the current version of the BRTMS.

The use of our DTMS will constrain the autonomy of an agent in one important aspect: Only one agent is allowed to be active at one time. There is no provision for two agents to change their beliefs simultaneously. Handling such cases is not trivial and is a subject for further research.

Bibliography

[AC89] R. M. Adler and B. H. Cottman. A development framework for distributed artificial intelligence. In *Proc. of the Fifth Conference on Artificial Intelligence Applications CAIA*, pages 115–121, Miami, FL, 1989.

[BG88] A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence.* Kaufmann, San Mateo, CA, 1988.

[BGB87] A. L. Brown, D. E. Gaucas, and D. Benanav. An algebraic foundation for truth maintenance. In *Proc. of the 10 th IJCAI*, pages 973–980, Milan, Italy, 1987.

[BH90] D. M. Bridgeland and M. N. Huhns. Distributed truth maintenance. In *Proc. of AAAI-90*, pages 72–77, Boston, MA, 1990.

[CG88] R. P. Carasik and C. E. Grantham. A case study of cscw in a dispersed organization. In *Proc. of CHI-88*, pages 61–66, 1988.

[DdK88] M. Dixon and J. de Kleer. Massively parallel assumption-based truth maintenance. In *Proc. of AAAI-88*, pages 199–204, St. Paul, MN, 1988.

[dK86] J. de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:125–224, 1986.

[dK90a] J. de Kleer. Exploiting locality in a truth maintenance system. In *Proc. of AAAI-90*, pages 264–271, Boston, MA, 1990.

[dK90b] J. de Kleer. A practical clause management system. SSL Paper P88-00140, 1990.

[DLC89] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. Trends in cooperative distributed problem solving. In *Transactions on Knowledge and Data Engineering*, 1989.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[EBMT89] S. F. Ehrlich, T. Bikson, W. Mackay, and J. C. Tang. Tools for supporting cooperative work near and far: Highlights from the cscw conference (panel). In *Proc. of CHI-89*, pages 353–356, 1989.

[Elk90] C. Elkan. A rational reconstruction of nonmonotonic truth maintenance systems (research note). *Artificial Intelligence*, 43:219–234, 1990.

[Her88] L. E. C. Hern. On distributed artificial intelligence. *The Knowledge Engineering Review*, 3:21–57, 1988.

[JK90] U. Junker and K. Konolige. Computing the extensions of autoepistemic and default logics with a truth maintenance system. In *Proc. of AAAI-90*, pages 278–283, Boston, MA, 1990.

[Kon87] K. Konolige. On the relation between default and autoepistemic logic. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 195–226. Kaufmann, Los Altos, CA, 1987.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming.* Springer, 1984.

[McA90] D. McAllester. Truth maintenance. In *Proc. of AAAI-90*, pages 1109–1116, Boston, MA, 1990.

[McC80] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 171–172, 1980.

[MH69] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[MJ89] Cindy L. Mason and Rowland R. Johnson. DATMS: A Framework for Distributed Assumption-Based Reasoning. pages 293–318, 1989.

[Moo85] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[Mor87] P. Morris. A truth maintenance based approach to the frame problem. In F. M. Brown, editor, *The Frame Problem in Artificial Intelligence*, pages 297–307. Kaufmann, Los Altos, CA, 1987.

[Mor88] P. H. Morris. Autoepistemic stable closures and contradiction resolution. In M. Reinfrank, J. de Kleer, M. L. Ginsberg, and E. Sandewall, editors, *Non-Monotonic Reasoning: 2nd International Workshop, Grassau, Germany*, pages 60–73. Springer, Berlin, Heidelberg, 1988.

[PZM90] I. Popchev, N. Zlatareva, and M. Mircheva. A truth maintenance theory: An alternative approach. In *Proc. of the 9 th ECAI*, pages 509–514, Stockholm, 1990.

[RDB89] M. Reinfrank, O. Dressler, and G. Brewka. On the relation between truth maintenance and autoepistemic logic. In *Proc. of the 11 th IJCAI*, pages 1206–1212, Detroit, MI, 1989.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Rei89] M. Reinfrank. *Fundamentals and Logic Foundations of Truth Maintenance.* PhD thesis, Linköping University, 1989.

[Rus85] D. Russinoff. An algorithm for truth maintenance. Technical Report AI-062-85, MCC, 1985.

[SMH90] Donald Steiner, Dirk Mahling, and Hans Haugeneder. Human computer cooperative work. In M. Huhns, editor, *Proc. of the 10th International Workshop on Distributed Artificial Intelligence*, 1990.

[Sou90] R. W. Southwick. The lemma generation problem. Imperial College, London, 1990.

[YHS88] J.-Y. D. Yang, M. N. Huhns, and L. M. Stephens. An architecture for control and communications in distributed artificial intelligence systems. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 606–616. Kaufmann, San Mateo, CA, 1988.

I would like to thank Prof. M. Richter and Project KIK of the DFKI for providing the environment conducive to this research. D. Steiner and A. Burt read numerous drafts of this thesis and made valuable comments. D. Mahling suggested ways in which a TMS could fit in with the global picture. Finally, I would like to thank my parents for giving me the necessary encouragement throughout my studies and work on this thesis.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-90-01
Franz Baader: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-02
Hans-Jürgen Bürckert: A Resolution Principle for Clauses with Constraints
25 pages

RR-90-03
Andreas Dengel, Nelson M. Mattos: Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder, Werner Nutt: Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader: A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist: Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:
1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo: Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann: Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist: Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley: Declarative Operations on Nets
43 pages

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-90-17
Stephan Busemann
Generalisierte Phasenstrukturgrammatiken und ihre Verwendung zur maschinellen Sprachverarbeitung
114 Seiten

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka :
On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt:
The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-04
Harald Trost
X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents
A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist
WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta
RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke
A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel
Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-13
Gert Smolka
Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-15
Bernhard Nebel, Gert Smolka
Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann
Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel & Nelson M. Mattos
The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

DFKI Technical Memos

TM-89-01
Susan Holbach-Weber: Connectionist Models and Figurative Speech
27 pages

TM-90-01
Som Bandyopadhyay: Towards an Understanding of Coherence in Multimodal Discourse
18 pages

TM-90-02
Jay C. Weber: The Myth of Domain-Independent Persistence
18 pages

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler
Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann
Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt
Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel and Rainer Bleisinger
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

DFKI Documents

D-89-01
Michael H. Malburg, Rainer Bleisinger: HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

D-90-01
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

D-90-02
Georg Seul: Logisches Programmieren mit Feature -Typen
107 Seiten

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten

D-91-01
Werner Stein , Michael Sintek
Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause
RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-06
Gerd Kamp
Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause
Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 pages

D-91-09
David Powers and Lary Reeker (Eds)
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.)
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
Thilo C. Horstmann
Distributed Truth Maintenance
61 pages

Distributed Truth Maintenance

Thilo C. Horstmann

D-91-11
Document