Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Technical Memo
TM-92-02

Organizing Communication and Introspection in a Multi-Agent Blocksworld Scenario

Achim Schupeta

March 1992

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

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

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
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 Daimler-Benz, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Digital-Kienzle, Philips, Sema Group Systems, 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

Organizing Communication and Introspection in a Multi-Agent Blocksworld Scenario

Achim Schupeta

DFKI-TM-92-02

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-9104).

ORGANIZING COMMUNICATION AND INTROSPECTION IN A MULTI-AGENT BLOCKSWORLD SCENARIO

Achim SCHUPETA

DFKI (German Research Center for Artificial Intelligence)
Research group AKA-MOD
Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
GERMANY

email: schupeta@dfki.uni-sb.de

The implementation of a simple blocksworld-scenario simulation-program is described. The blocksworld is modeled according to the multi-agent paradigm of distributed artificial intelligence. Each block is viewed as an agent. The agents have capabilities like to move, to communicate, to plan or to gain a small ammount of introspective knowledge which are necessary to transform the initial scene of a problem into the goal scene. The structure of the system is oriented along the ideas of the specification of RATMAN described in (BMS91). RATMAN was reduced to its two central modules and their concepts were implemented with means as simple as possible. The result was a system, that allows to experimentally develop concepts for communication, planning and introspection, that are (for this simple toy-domain) sufficient to solve the problems without any global problem solver, but by the cooperative behavior in the society of agents

Contents

1. Introduction

In the last years there was a trend in nearly all disciplines of computer science towards the distribution of systems. Maybe, that this trend was initiated by the development in the area of the hardware, where huge mainframe computers were pushed away by networks of several small computers in many application domains. The virtue of distributed systems lies in their reliability. If in a centralized system a failure occurs, the whole system is going down. In a distributed system, the tasks of a faulty component may be (at least partially) performed by other components, and even in the case of several failures, the system can degrade gracefully.

Another advantage of distribution is a better management of complexity. In a huge central system even simple modifications or extensions will have major impacts to the system. Distributed systems on the other hand are usually designed in a process of incremental extensions and thus will be more flexible.

In AI the idea of distribution led to the research area of DAI, which is concerned with *concurrency* in AI at many levels. DAI branches into the area of *distributed problem solving* (DPS), which considers how the work of solving a particular problem can be divided among a number of cooperating and knowledge-sharing modules or nodes, on the one hand, and into *Multi-Agent Systems* (MAS) on the other hand. In MAS the coordination of intelligent behavior of a collection of autonomous intelligent agents is the main concern. Coordination of knowledge, plans, different skills and actions itself is a process, that the agents have actively to reason about. An overview of DAI is given in (Sch92) and a collection of prominent research papers can be found in (BG88).

In the research projekt AKA-MOD at the DFKI we are interested in the paradigm of *autonomous cooperating agents*. How can we simulate a society of agents such that the cooperative problem solving behavior in human society is rebuilt? What motivates the agents to cooperate? Which role plays communication is such a society, and what level of communication is appropriate? In DAI the interest in distributed systems is much deeper motivated than in gaining more relaible systems or a better management of complexity: Consider that if the reductionistic viewpoint, that a society is built by its agents, could be enriched with the reverse viewpoint, that the agents are formed by the society, a deep insight into the intelligence of the agents would be gained by understanding the concepts of interaction and cooperation at the level of the society. AI always should take a look to natural intelligence and it is hard to believe, that natural intelligence would have been developed without a society. This aspect of intelligence was always neglected in classical AI.

In the AKA-MOD project we aim to do first steps towards an agent-society

first by designing a comfortable testbed to perform experiments in the modelling and simulation of agent-societies in general and

second by taking a closer look to a few concrete application domains in which a cooperative problem solving behavior of several agents promises fruitful results.

The first goal already led to the specification of RATMAN, which stands for "Rational Agents Testbed for Multi-Agent Networks" and is described in (BMS91). For the second goal, the following suggestions were made:

The problem of the towers of hanoi, where each disk is modeled as an agent.

A simple blocksworld scenario, where each agent represents a block.

A loading dock, where agents correspond to forklifts, cranes, trucks and so on.

Several transportation companies, that compete for orders, but also cooperate in performing orders that are too large for a single companies' resources

The complexity increases from toy-world examples to real world domains of remarkable complexity. The towers of hanoi and the blocksworld scenario are already implemented as simple experimental simulation programs. The loading dock and the transportation company scenarios are still in a specificational state.

The purpose of this paper is to report about the implementation of the multi-agent blocksworld scenario simulation. A few remarks seem to be necessary before going into the details:

1.) The simulation program tries to implement the main ideas of the RATMAN-specification, which will briefly be reviewed in the second section. Nevertheless we tried to implement those ideas with means as simple as possible to invest not too much work into this toy-world scenario.

2.) PROLOG was used for the implementation, because it seems to fit best in our aim to stay within the logical paradigm on the one hand. PROLOG programs consist of Horn clauses and thus of logical formulas. On the other hand PROLOG programs also describe algorithms. PROLOG is therefore as ideal to express *knowledge* as it is to express *skills*. In particular RATMAN's conceptualization of an agent is that of a hierarchical structured knowledge base and therefore a structured PROLOG program is a good and simple first approach to that concept.

3.) The concurrent behavior of the different agents is only simulated! The top-level loop of the program calls the agents sequentially, each agent performs some simple actions and returns the control back to the loop. This liberates the simulation from various kinds of synchronization problems that will occur in more sophisticated simulations of multi-agent systems, where it is planned to associate different processes on different computers with the different agents.

4.) The user interface of the system is not very comfortable. One can give an initial scene and a goal scene during a simple dialogue. Then the system initializes the agents with their local goals. The user can see the exchanged messages and information about their intentions, plans and introspective knowledge. A pictoral representation of the performed movements can be seen after the run of the system.

2. Structure of the System

In the following we present an overview of the structure of the blocksworld simulation program. The detailed description of the presented modules will follow in the related sections below. The agents are seen as hierarchically layered knowledge bases, where each layer represents certain skills of the agent. This idea is taken from the specification of RATMAN, that will shortly be reviewed first:

2.1. RATMAN: A Specification

In this section we will present a brief review of a proposed testbed for multi-agent systems RATMAN, which stands for Rational Agents Testbed for Multi Agent Networks. For a more detailed description see (BSM91). RATMAN conceptualizes a universal testbed along four main modules, which are the agent tool box, the current world scenario, the specification kit, and the status sequence and statistics box:

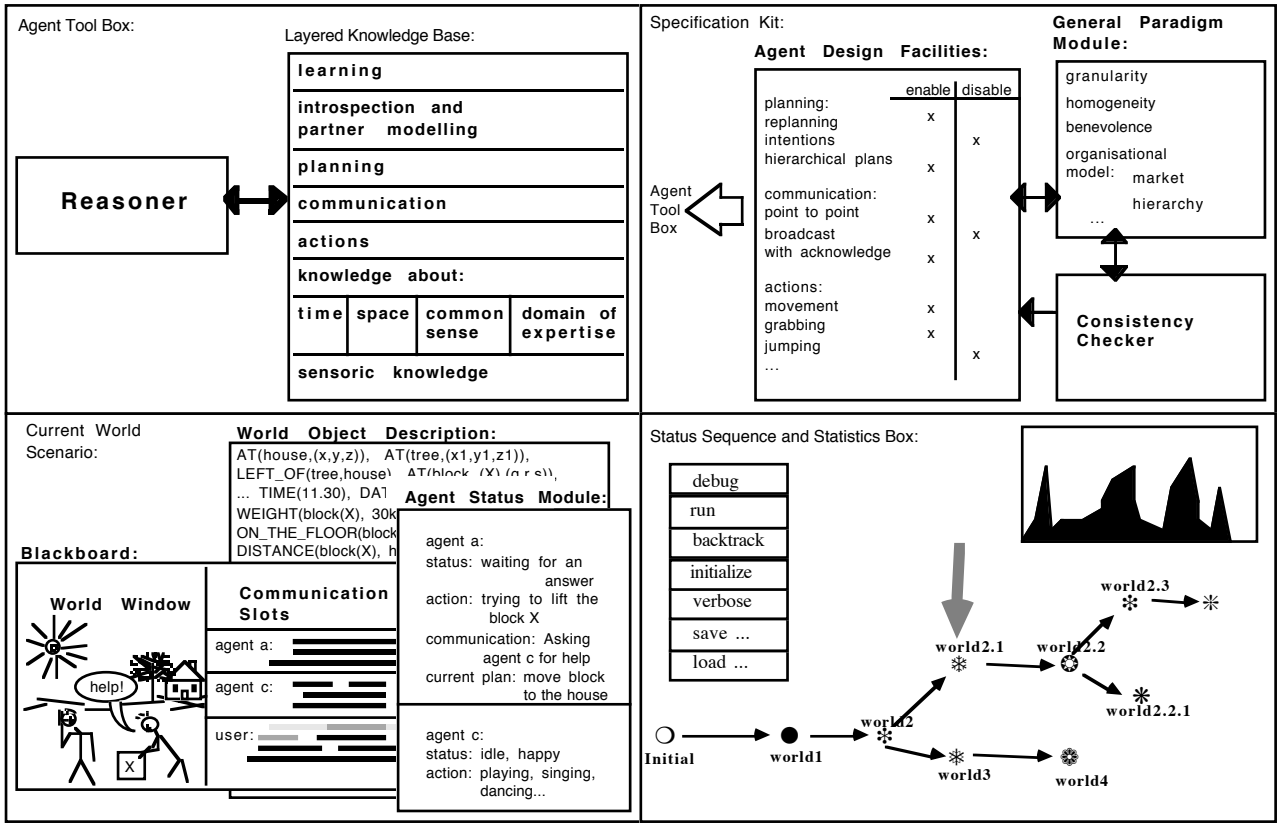
The **agent tool box** provides a scheme for a *hierarchically structured knowledge base* together with *reasoning facilities* to model all features of agents. Predefined knowledge may be used or be partially skipped and new knowledge may be introduced by the user. The intention is to develop a library of several knowledge packets for the different levels of the hierarchy with various possibilities of combinations. The different levels of the knowledge base will be responsible for different types or aspects of an agent's behavior. For example there could be layers for sensoric knowledge, knowledge about time, space, common sense or expert-knowledge, knowledge about actions, communication, planning schemes, introspective knowledge and learning capabilities.

Each agent defined in the agent tool box will be given a place in the **current world scenario**. It contains a large knowledge base that describes all *world objects* and from which only a part is visible to the user. Furthermore there is the *agents status module*, that presents the agents internal status of the agents to the user and last not least the *blackboard*, which serves as a communication platform. The blackboard itself is split into a communicational part, which provides *communication ports* for each agent (and for the user) and a *world window*, where all objects and agents in the scene are represented.

The **specification kit** serves as the user interface to *define the agents*, their relations in the world, and their status in the agent society. It will provide several choices for general strategies to be performed and it can be used to specify what kind of status information and statistic data should be monitored by the system.

The **status sequence and statistics box** has the task to *show the sequence of the changing world states*, the activity potential of the agents, the internal clock, analysis of synchronization processes etc. Depending on the specification of the actual scenario, a great amount of specific information should be accessible to the user via this module. Also *tracing* and *debugging facilities* on an appropriate level should be provided.

The following figure illustrates this structure:

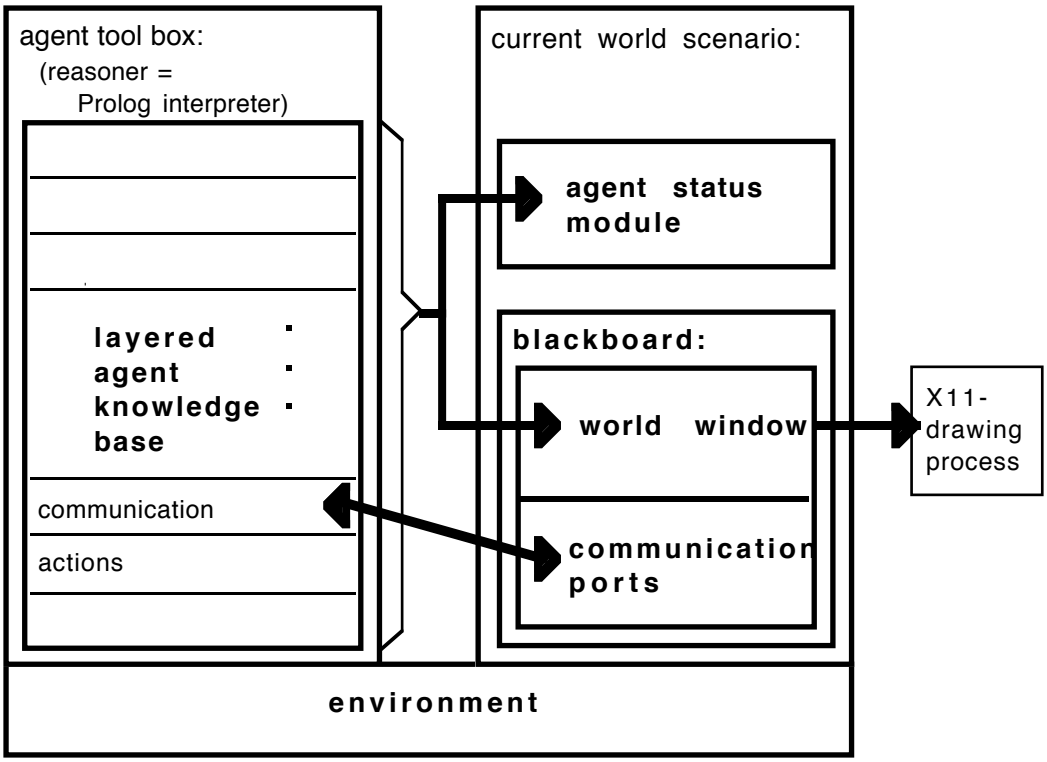


2.2. Mapping into an Implementation

As already mentioned, the simple multi-agent blocksworld simulation program tries to implement RATMAN's central ideas by means as simple as possible. It consists of restricted simple versions of the two more important modules of RATMAN, which are the *current world scenario* and the *agent tool box*.

The specification kit's function, which is to support a high level interface for the agent design and control and check the agent generation, is not implemented in the cuurent version of the system. Also the status sequence and statistics box is completely left out.

The following picture sketches how restricted versions of these two modules are organized in the blocksworld simulation program. The following broad description of the components of this picture shall give an overview of the system structure. A more detailed description can be found in the related subsections:



What is called ***environment*** in the figure is a collection of all those parts of the program, that have nothing to do with a single agent and cannot be attached to the components of the current world scenario. Examples are the predicates that realize the *agent-activation-cycle* (= top-level program), *initialize* the scene, invoke the *drawing routines* and several subroutines of minor interest. The environment or more exactly the agent-activation-cycle acts like a controlling program, that invokes the agents alternating, until all agents have reached a status, where all their goals are reached.

The ***agent tool box*** is simplified by the following idea: The *layered knowledge base* is just a structured PROLOG program. The *reasoner* is then provided by the interpretation algorithm of PROLOG. The contents of the different layers will be described in section 4, which focuses especially on the aspects of action, communication, simple planning and simple introspection.

The ***current world scenario*** has no world object database in this implementation, because the objects are the agents themselves and they know their own positions. Other world knowledge does not exist or more precisely is too rare to justify a separated module. For example, the world knowledge about the x- and y-range of the scene and the position of the supporting table on which the blocks stand is hidden in the common sense knowledge layer of the agents.

The ***agent status module*** provides various information about the agents' internal state. It is a form of filter on parts of the agents' knowledge bases. For example it is shown what the agents communicate, what their actual plans and intentions are, what their actual introspective knowledge is and so on.

The ***blackboard*** should not be confounded with the central shared data-structure of a typical blackboard-architecture. In our first simple blocks world scenario, the communicational part of the blackboard is just used as a device to implement ***communication channels*** between the agents. The ***world window*** part of the blackboard has a similar task as the agent's status module, but shows the world objects instead of the agents' internal status.

To attain a visible picture of the scene, the world window is supplied with drawing routines, that generate commands for a ***drawing process***, which draws a scene picture using the X Windows system.

3. Description of the Scenario: The Blocksworld

Blocks world scenarios are frequently used in literature about planning systems. Usually the blocks world is composed of a *table*, several *labeled blocks* and a movable *robot-arm*, which is able to grab one block at a time, to move it to another place and to put it down. Each block can directly stand on the table or on another block, which may be supported by the table or a third block and so on. The goal is to generate a plan composed of a sequence of simple operations, that transforms an initial scene into a given goal scene. The *description of a particular scene* can be done with a few simple predicates:

ONTABLE(X)	: Block X is standing on the table.
ON(Y, X)	: Block Y is standing directly on block X.
CLEAR(Y)	: No block stands on block Y.
HANDEMPTY	: The robot-arm is empty.
HOLDING(Z)	: The robot-arm is actually holding the block Z.

A scene thus is described as a consistent conjunction of these predicates. Such a description is of course only an abstraction, because several details are not represented. For example ONTABLE(A) & ONTABLE(B) does not make any assertion about which block is right and which is left of the other. *Basic operations* that can be performed are:

pickup(X)	: The robot-arm grabs block X from the table.
putdown(X)	: The robot-arm lays block X down on the table.
unstack(X, Y)	: The robot-arm grabs block X from block Y.
stack(X, Y)	: The robot-arm lays block X down on the block Y.

Again the operations are abstractions, that leave out irrelevant details and make several implicit assumption about the robots capabilities. To perform a stack-operation for example the robot must know or perceive the exact position of the supporting block. Several different planning algorithms and modified representations of operators and scenes around this blocks world scenario have been discussed and analyzed in the literature. For examples see (Cha87) (Wal75).

To adapt the blocks world to a scenario, that involves *more than one agent*, we can think of two possibilities:

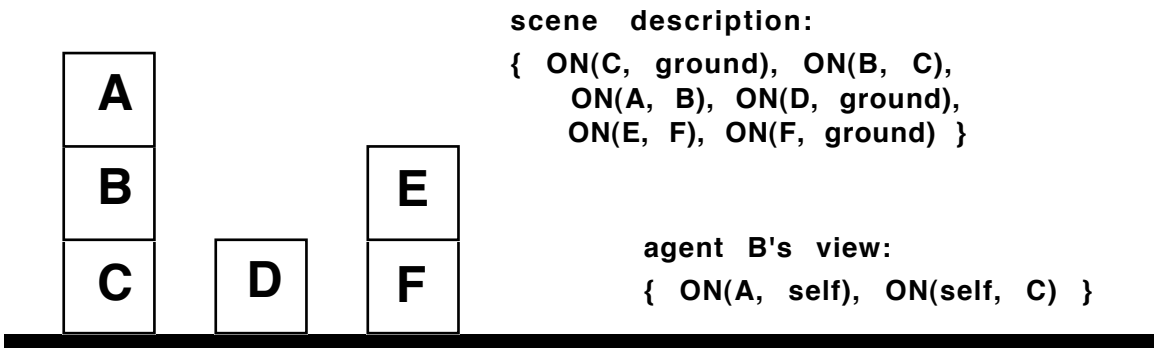
- First more than one robot-arm can be introduced. Each agent controls one arm and the agents have to coordinate their actions. The coordination in a simple scenario will be restricted to the avoidance of harmful interaction e.g. trying to grab the same block at the same time.
- The second possibility was inspired by the idea of the eco-problem solving paradigm. (confer (Fer90)). The intelligence is assigned to the blocks themselves. Each *block is modelled as an agent*, that only aims to fulfil its own local goals and even did not know about the global goal. This modelling sounds very much like a reactive behavior approach, but each agent can be equipped with more than only reactive intelligence, for example simple planning and communicative capabilities and consciousness about the (local) problem solving state. So this modelling comprises effects of reactive approaches (global solution as an emergent phenomenon of the local problem solving behavior) as well as the aspects, that are typical for the logic based approach.

We decided to use the second possibility. The interesting questions are: How far would the local orientation of the agents reach? Which level of complexity would require a global view and how can an agent gain this global view?

The following modifications result from our modelling:

- The predicates HOLDING and HANDEMPTY are superfluous, because since the blocks are autonomous agents in our representation, there is no robot-arm anymore.
- A global scene description is only necessary for the definition of the initial scene and the goal scene.
- The predicate ONTABLE(X) is represented by an ON(X, ground) for purposes of a simple and uniform representation, though we do not model the 'ground' (=table) as an autonomous agent.
- The predicate CLEAR(X) is also left out in the global scene description: Block X is clear if there is no other block Y such that ON(Y, X) holds.

Each agent has to figure out by communication if there is another agent located above it before it moves and has to announce every own movement to the others, so that they can maintain their knowledge. Thus the predicate CLEAR(X) is internalized into the agents private knowledge bases. A scene description is now reduced to a set of ON-relations. The following figure gives an example:



The agent's view of the goal-scene is restricted to those relations, that it is participated in. An agent's goal can only be to be ON another agent (or on the ground) or to be below another agent.

The basic operations of the blocks world are generalized and condensed into one abstract operation named **move_on(X)**, which is intended to represent the capability of an agent to *jump* on another block or on the ground. If X is an agent, the acting agent is assumed to have knowledge about the exact location and size of X and if X is 'ground', the agent has to search for a free place on the table. We assume that there are no limitations for such a jump relative to height and width. Other actions of the agents like communication or changes of their internal states are not visible directly in the picture of the scene and will be described in more detail in the next section.

4. The Agents

We see an agent as a hierarchically layered knowledge base, which is implemented as a structured PROLOG program. Fortunately the agents in our blocks world scenario are all similar. Differences in the size and position and the internal states occur, but in principle we have *homogeneous agents*. Therefore nearly all predicates are provided with an *agent-parameter*. In the predicates that hold for all agents, this parameter is filled with a variable, while in those predicates that hold for a specific agent the agent name is used.

In the following section we'll describe the principles of our blocks world agents' behavior. Then the next section tries to associate the different skills of an agent to the layers, that are suggested in the description of the agent tool box of RATMAN. The layers in focus of our interest, which are action, communication, planning and introspection & partner modelling, are described in detail in the then following sections.

4.1. The Agents' Behavior in General

Our conceptual considerations led to the **agent-activation-cycle**: The environmental loop sequentially activates all agents of the scenario with a call of the predicate activity(X), where X is the name of the actual activated agent. This predicate is therefore

an important part of the interface between an agent and the environment. The environment gives control to the agent. Now we must fix what an agent does within an activation.

The **general paradigm of activation** is the following: The activated agent looks for messages in its communication slot of the blackboard and processes them in dependence to their contents. Then the agent performs actions according to its internal state and returns the control to the environment. The actions according to the internal state are directed to satisfy the agent's intentions.

The **general paradigm of planning and intention** is: Each agent has a list of intentions, that it tries to achieve one after the other. The ***actual intention*** is used to generate a ***plan*** which is a sequence of ***basic plan-steps***, also called ***goals***. A basic plan-step is resolved into ***simple actions*** like for example a change in the agent's internal knowledge base, asking a question to another agent, modifying the own plan by insertion of a plan-step or jumping on another agent. The actions performed in a single activation usually correspond to the execution of a single basic plan-step. For example after sending a question the receiver agent needs an activation to have a chance to generate an answer.

The **general principles of communication** are: Our agents send their messages to specific other agents or broadcast them to everyone. The level of communication is relatively primitive. A finite set of "***communication tokens***" comprising all possible information exchange demands that may occur in our scenario is predesigned. The tokens are clustered to certain types like ***question, answer, announcement, instruction***. A message consists of the names of the sending and the receiving agents, a message type identifier and the message token which may bear a few parameters.

The **general principle of locality** says that an agent only interacts with those agents that it knows. In the beginning of a scenario the agents *become acquainted* with those agents, that occur in their intentions. (These are their neighbors in the goal scene.) They may become acquainted with other agents later on if they need to communicate with them.

The **general principle of private knowledge** refers to the agent's capability to maintain a private knowledge base. Real ***introspective knowledge*** - like answers to the questions: Do my intentions imply a future movement of me? Do I actually have a plan to perform? Are all my intentions fulfilled? - is mixed with knowledge about the actual ***spatial relations*** in the scenario (Is someone on me? Is someone on agent X? What is the position and the size of agent Y?) and knowledge that is simply used to ***remember*** some actions in the past. For example if someone replies a "yes" to you, you have to be aware of the question that you asked him just before. We also use private knowledge to *decouple communicational acts from getting aware of the communicated information*. For example an agent wants to know if another agent has an intention to move. It first inspects its private knowledge base if it already knows something about that. If not, it sends a question to the other

agent. The reply message is used to update the private knowledge in that point. Then the same action as in the activation before takes place, that is the agent searches again in its knowledge base but this time is aware of the other agent's intention because of its updated knowledge base.

4.2. The Different Layers contain the Agents' Abilities

The attachment of the predicates that realize an agent to the different layers turned out to be not so easy. In this sense our chosen attachment is not too rigid and open to other interpretations. The following table seemed an appropriate attachment to us:

layer	attached predicates and description
sensoric knowledge	**position** and **size** of each agent expressed in x,y-coordinates of the upper left corner of the agent and height and width. Each movement retracts the old position and asserts the new one.
temporal knowledge	----------
spatial knowledge	the **ON-relation** is used to test if one agent is on another or on the ground.
common sense knowledge[1]	knowledge about the y-level of the table and the maximal range of x- and y-coordinates.
expert knowledge	----------
actions	the **activity** predicate to be described in more details soon; a predicate that associates **simple actions** to execute to each basic plan-step; a few predicates that support the agent movement and the search for a free place on the table when the agent jumps on the ground.
communication	predicates to **send**, **receive** and **broadcast** messages; a predicate **process_message** that

[1]This layer is for sure the most contestable for firstly it is not clear what is comprised by common sense knowledge in general and how it is modelled and secondly in our scenario much common sense is implicit in the design. An example is the implicit expectation of each agent, that other agents have the same structure and will understand its messages. Therefore what we attached here to this layer is only a stop-gap solution, but we did not pay much attention to common sense in this simple scenario anyway.

	is used within the activity predicate to work on the received messages.
planning	the list of **intentions** for every agent; the **actual intention** that each agent is working on; the actual **plan** generated from the actual intention; a predicate **remember** to store already fulfilled intentions; a predicate to **generate a plan** from an intention.
introspection & partner modelling	the **iknow** predicate, that realizes each agent's **private knowledge** base; a predicate that checks the actual plan and all unfulfilled intentions if they will force the agent to do a movement in the future.
learner	----------

We see that a few layers are yet not used in our simple blocksworld scenario or are only sparsely filled. Our special interest was focused to the levels of action, communication, planning, introspection and partner modelling. We'll have a closer look at these layers in the following sections. We are now going very much into the details of the implementation and even program code will occur in the next sections.

4.3. Introspection & Partner Modelling

A crucial question in our blocks world is if an agent will move in the future. If an agent intends to jump on agent X, but agent X is still not at its final position, i.e. will move in the future, it is better to wait until agent X has moved because otherwise the agent will have to jump down from X again because a precondition for agent X to move is that no one is located on it. We do not represent an **intention to move** explicitly, so a special procedure checks the actual intention, the intentions and the actual plan for such an intention to move. To get an impression how clear and straightforward this is realized in PROLOG, we show a cut from the code that realizes this procedure[2]:

```
%*** introspection into the own movement-intentions:
intention_to_move(Agent) :-
                    act_intention(Agent, on(self, X)),
                    (on(Agent, X) ->     fail | true).

intention_to_move(Agent) :-
                    intention(Agent,L),
```

[2]The conditional goal construction "A -> B | C " in Quintus PROLOG stands for " if A then B else C"

```
                member(on(self, X), L),
                (on(Agent, X) ->    fail
                                |
                                    true).
intention_to_move(Agent) :-
                plan(Agent, L),
                member(move_on(X), L),
                (on(Agent, X) ->    fail
                                |
                                    true).
```

The other part of this layer are the **private knowledge bases** of the agents. They expand or shrink dynamically during the run of a scenario, because the agents continually assert or retract knowledge. The predicate iknow is parameterized by the agent's name and a **knowledge token** representing a fact. The intended meaning is that the agent knows this fact. All existing knowledge tokens of our scenario are listed and shortly described in the following:

➪ Knowledge about other agents:

iknow(A, [existence, X])	:	A knows about the existence of agent X.
iknow(A, [pos-size, X])	:	A knows the position and size of agent X.
iknow(A, [is-fixed, X])	:	A knows that X will not move in future.
iknow(A, [intends-to-move, X])	:	A knows that X intends to move.
iknow(A, [has-moved,X])	:	A knows that X has moved.
iknow(A, [is-free, X])	:	A knows that no other agent is on X.
iknow(A, [not-free, X, under,Y])	:	A knows that Y is on X.

➪ Introspective knowledge:

iknow(A, [i-have-a-plan])	:	A knows that it actually has a plan to perform.
iknow(A, [no-more-intentions])	:	A knows that it has no more unfulfilled intentions.
iknow(A, [act-intention-fulfilled])	:	A knows that its actual intention is fulfilled.
iknow(A, [i-am-fixed])	:	A knows that it has no further move-intention.
iknow(A, [i-intend-to-move])	:	A knows that it has a further move-intention.
iknow(A, [i-am-free])	:	A knows that no other agent is on it.
iknow(A, [is-on-me, X])	:	A knows that agent X is on it.

➪ Awareness of previous actions:

iknow(A, [asked-movement, X])	:	A knows that it has asked X for its move-intention.
iknow(A, [asked-freedom])	:	A knows that it has broadcasted a question about its freedom, i.e. if someone is on it.
iknow(A, [asked-freedom-of, X])	:	A knows that it has asked X weather someone is on X.

4.4. Planning

The planning layer bears the **intentions**, the **actual intentions**, the **plans** and a **remember** facility for each agent. In the initialization of a scenario the user has to specify an initial and a final scene. From the final scene the environment computes each agent 's intentions and stores them as facts of the intention predicate. For example a final scene as depicted in the figure of section 3. would lead to the following intentions:

```
%*** planning-layer: ********************************************************
%*** intentions:
intention(a, [on(self, b)]).
intention(b, [on(self, c), on(a, self)]).
intention(c, [on(b, self), on(self, ground)]).
intention(d, [on(self, ground)]).
intention(e, [on(self, f)]).
intention(f, [on(e, self), on(self, ground)]).
```

The actual intention in each case is the first intention of the list. A fulfilled intention is stored in a similar list with another predicate called **remember**. This is necessary because it is possible in some situations, that an agent has reached its satisfaction state (i.e. all its intentions are fulfilled) but these intentions only encode the agent's local view and the global goal may require the agent to move away once more. In that case the agent must have the capability to remember its original intentions after its movement.

Plans are generated from the actual intention as shown in the following code segment:

```
%*** plan generation:
generate_plan(Agent, Plan) :-
                    act_intention(Agent, on(self, X)),
                    Plan = [know(X), test-intention, fix(X), free(X), free(self), move_on(X)].
generate_plan(Agent, Plan) :-
                    Plan = [know(X), test-intention, free(self), fix(self), wait_for_move(X)].
```

We see that the intentions are very rigidly transformed to sequences of **simple plan steps** or goals. For example the intention on(self, X): The last goal is to move on agent X. The other goals are to assure the **preconditions** of that movement: The agent must know the agent X, it must test if it is not already on X, it must be sure, that X has not the intention to move in future, that no other agent is already on X and that no other agent is standing on itself. Comparing this with the STRIPS representation of operators (cf.(FN71)), we miss the delete- and add-list, used to describe the **effects of an operation** by specifying how to maintain the knowledge base when applying the operator. In our scenario, the goal-achievement actions for move_on(X) will change the position of the moving agent by assert and retract operations and broadcast an announcement message to all other agents so that they can update their private knowledge bases. Therefore the effects of an operation are hidden in the operation itself. A list of all simple goals and the description of their intended meaning follows:

know(X)	Get acquainted with X if you are not already. That means to know about the existence, the size and position of X.

test-intention	Test if the actual intention is already fulfilled.
pause	Causes the agent to do nothing during its next activation. This goal is never used in the plan generation but inserted afterwards into plans to modify them for synchronization purposes.
answer-freedom	Focuses the agent's activity to the fact, that it has broadcasted a message “[question, someone-on-me, [x, y], len]” (confer communication layer) to all other agents. The agent has to keep the awareness of this fact on the plan level because it may possibly get no reply message at all and has to interpret that case appropriately.
move_on(X)	Jump on agent X or on the ground if X is evaluated to 'ground'.
fix(X)	Make sure, that agent X is at its final position and has no further move-intention.
free(X)	Make sure, that no agent is located on X.
wait_for_move(X)	Wait until agent X has moved.

4.5. Communication

The basic communication skills are message **sending**, **receiving** and **broadcasting**. The realization of the communication ports on the blackboard is simply done by a predicate named blackboard. Sending a message means assertion of a fact of this predicate, which contains the **sender**, the **addressee** and the **message token**. The receipt of a message is conversely the retraction of a fact. The broadcasting capability should resemble a shouting behavior - everyone hears the message and anyone may reply to it. A totally new acquaintance can therefore only be installed with the help of a broadcast, because to send a message directly, the agent must already know the addressee, and from the beginning it only knows the names of its direct neighbors in the goal scene. The following code realizes these basic skills:

```
%*** communication-layer: ***********************************************
%*** basic communication skills:
receive_message(Agent, From, Message) :-
                    retract(blackboard(From, Agent, Message)).
send_message(To, Self, Message) :-
                    assert(blackboard(Self, To, Message)).
broadcast_message(Agent, Mess) :-
                    agentlist(AgList),
                    member(To, AgList),
                    send_message(To, Agent, Mess),
                    fail.          % use backtracking to capture all agents of AgList.
```

Our communicational model is simple because we use a predesigned set of **communication tokens**. We grouped them according to their communicational function into four **message types**: **questions**, **answers**, **instructions** and **announcements**. We go through these message tokens and give a description of the intended meaning:

➪ Messages of type **instruction**:

[instruction, pause] : This causes the receiving agent to do nothing in its next activation. This is achieved be inserting the goal 'pause' to the front of the plan.

[instruction, move-away-from-me] : To reach the goal free(self) an agent can instruct the agent, that blocks it, to move away. This instruction may reactivate even an already satisfied agent.

➪ Message of type **announcement**:

[announcement, i-jump-on, X] : After a movement an agent has to broadcast this message to everyone, because they may be interested in this agent's movement and must update their private knowledge base.

➪ Messages of type **question**:

[question, where-are-you] : To get informed about an agent's position and size.

[question, someone-on-me, [x,y], len] : Asks if someone is located on the agent. The agent has to supply its position's coordinates and its length, because the agent that replies may not be acquainted with it and therefore must gain that information from the message. As we have seen in the previous section, there may be no reply at all to this question and then the agent can assume, that no other agent blocks it.

[question, you-intend-to-move] : To achieve the goal fix(X) an agent can ask X by sending this message to X.

[question, are-you-free] : To achieve the goal free(X) an agent can ask X by sending this message to X. It may happen that agent X for itself does not have this information and must broadcast a message 'someone-on-me' first to gain it before it can reply.

➪ Message of type **answer**:

[answer, i-am-at, X, Y, W, H]	: Answer to the question 'where-are-you'. The additional parameters are the X- and Y-coordinates and the width and height of the agent.
[answer, yes]	: Answer to several possible questions. Therefore the receiving agent must be aware of the question to which this answer refers.
[answer, no]	: Analog to the answer 'yes'.
[answer, no, on-me, X]	: Negative answer to the question 'are-you-free'. This time the name of the blocking agent is supplied, because the asking agent must know for whose agent's movement it must wait.

The largest part of the communication layer is occupied by the predicate '**process_message**', that implements a procedure which describes how to deal with the received messages. To each of the presented communication tokens, it associates in dependance of the private knowledge base of the agent certain reactions like answers in response to questions, plan modifications in response to instructions or updates of the private knowledge base in response to announcements. We do not want to describe the hole code of this procedure, but discuss a little interaction scenario, which involves several messages and gives also an example for the interaction of the different knowledge layers. Therefore we first present that part of the code of the predicate 'process_message', that will be helpful to follow the example presented afterwards:

```
%*** effect of messages:
%*** effect of messages of type question:
process_message(Agent, From, [question, someone-on-me, [XB, YB], Len]) :-
        position(Agent, [X, Y]),
        size(Agent, [W, H]),
        X >= XB,
        X+W =< XB+Len,
        Y+H =:= YB,
        send_message(From, Agent, [answer, yes]).
process_message(Agent, From, [question, are_you_free]) :-
        (iknow(Agent, [i-am-free]) ->
                send_message(From, Agent, [answer, yes])
        |
        iknow(Agent, [is-on-me, Y]) ->
                send_message(From, Agent, [answer, no, on-me, Y])
        |
        otherwise ->
                send_message(From, Agent, [instruction, pause]),
                position(Agent, [XB, YB]),
                size(Agent, [Len, _]),
                (broadcast_message(Agent, [question, someone-on-me, [XB, YB], Len]) ->
```

```
                                        true
                                |
                                        insert_to_plan(Agent, answer-freedom),
                                        insert_to_plan(Agent, pause),
                                        assert(iknow(Agent, [asked-freedom]))))).

    %*** effect of messages of type answer:
    process_message(Agent, From, [answer, yes]) :-
                    retract(iknow(Agent, [asked-freedom])),
                    assert(iknow(Agent, [is-on-me, From])),
                    insert_to_plan(Agent, know(From)).

    process_message(Agent, From, [answer, yes]) :-
                    retract(iknow(Agent, [asked-freedom-of, From])),
                    assert(iknow(Agent, [is-free, From])).

    process_message(Agent, From, [answer, no, on-me, Y]) :-
                    retract(iknow(Agent, [asked-freedom-of, From])),
                    assert(iknow(Agent, [not-free, From, is-under, Y])).

    %*** effect of messages of type instruction:
    process_message(Agent, From, [instruction, pause]) :-
                    iknow(Agent, [i-have-a-plan]),
                    insert_to_plan(Agent, pause).
```

As we have seen in the planning layer, one precondition for a movement is, that the agent on which an agent intends to move is **free**, i.e. no other agent is already standing on it. Therefore an agent A may send an agent B the **question message 'are-you-free'**. Agent B processes this message as we see from the code presented above in the following way: It determines whether it has any according positive or negative information in its private knowledge base. If this is the case, it sends back an answer message **'yes'** or an answer message **'no, on-me, Y'**, that is completed with the name Y of the agent above B immediately. Let's assume now that no information about that is available in B's private knowledge base, i.e. agent B does not know, whether it is free or blocked by someone. Then B sends the instruction **'pause'** to A and broadcasts the question **'someone-on-me'** provided with its own location's x,y-coordinates and its length to everyone in the scenario and modifies its plan appropriately by the **insertion of a new goal 'answer-freedom'**, which will make B aware of its question in the next activation. This is necessary because it is possible that no one replies to the question. Agent B also forces itself to do nothing within the actual activation, which is done again by modifying its own plan with the **insertion of the goal 'pause'**. To be able to determine the reference of a reply, B stores in its private knowledge base, that it just has broadcasted that question (i.e. it asserts **'iknow(B, [asked-freedom])'**). During the next activation cycle, agent A does nothing because it obeys B's instruction to pause, while B gets the result of its question: A 'yes' if an agent is on B or no answer otherwise. B will therefore become aware of its situation, i.e. it stores in its knowledge base whether it is free or blocked. Finally the third considered activation now proceeds like the first

one; that means A asks again, but this time B has information about its freedom in its knowledge base and is able to reply a positive or negative answer.

One first result, the design of the communication level has taught us is that even in a simple scenario the complexity of communicational interaction is not trivial. We had to deal with **three different types of questions: First** a question where the answer is definite, i.e. the agent would understand the answer even if it never would have posted the question. For example the [answer, i-am-at, X, Y, W, H] will have the same effect for the receiver if it is the reply to a previous question or if no previous question exists. **Second** a question that the agent has to remember, because the possible answers are also possible answers to other questions. Examples for this are, 'yes' and 'no'. The **third** type has to take into account, that no answer is also an answer: This is the case for the [question, someone-on-me]. Only an agent at the right position would reply a 'yes' to that message, but if there is no answer at all, the asking agent has to interpret the missing of answers as the fact that no agent is on it. In our activation cycle model this is relatively easy to implement because we can guarantee the receipt of an answer one activation after the question was sent. In general this question type will introduce the problem of delay times due to transmission-time of messages.

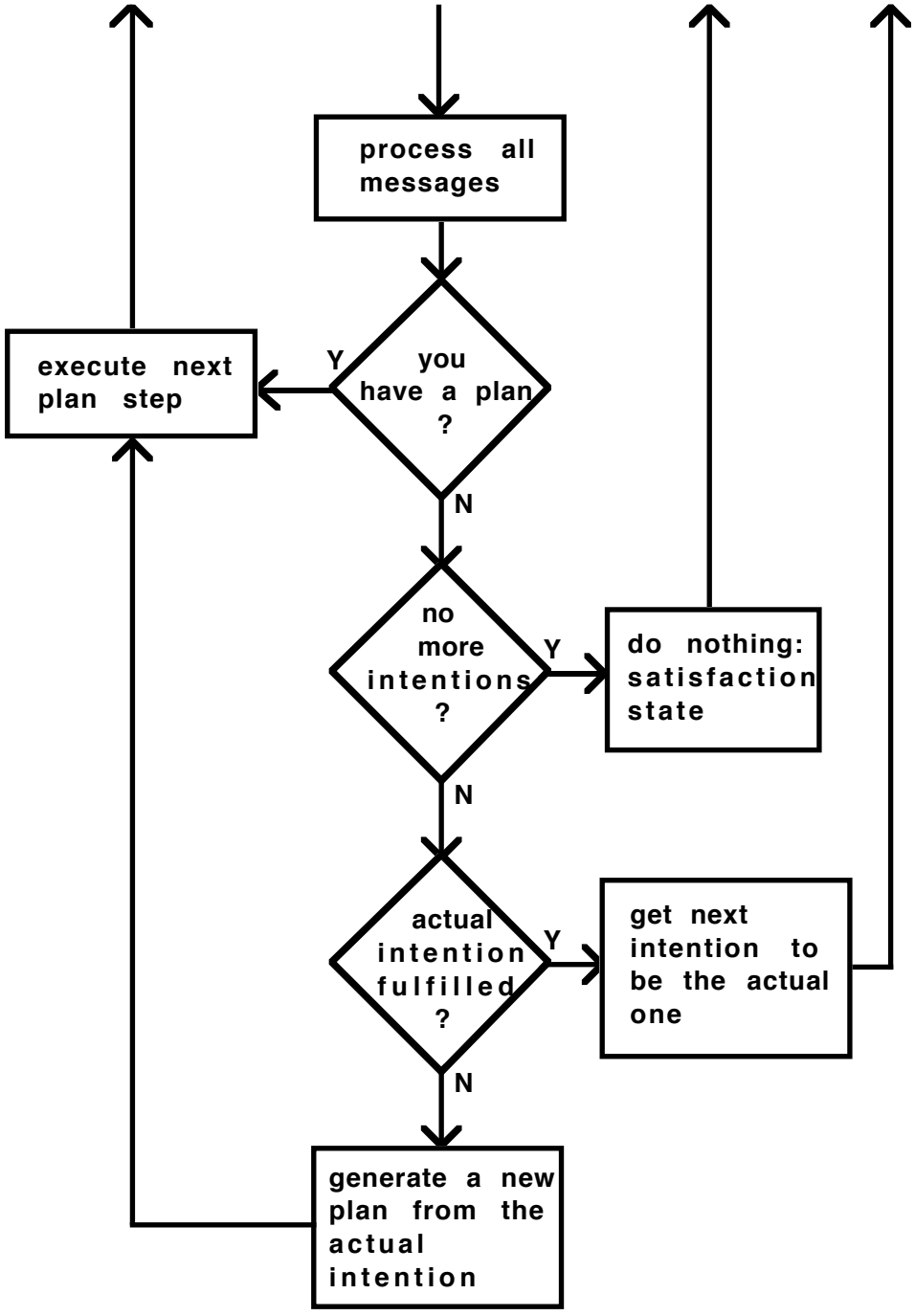
4.6. Actions

The action layer has two major components: The **activity-function**, which is the entry point to every activity performed by an agent and a predicate **'reach_goal'**, that associates simple actions to each goal and executes them. The activity-function proceeds according to the following **rules**:

- **Look for messages** in your communication port of the blackboard and process them. This is done by the predicate 'process_message' already discussed in the previous section and may include sending of responses, modifications of the plan and even the intentions and updates of the private knowledge base.
- If you have a plan, then **execute the next plan step**. The plan is processed sequentially and thus always the first plan step is taken to be executed by a call of the predicate 'reach_goal' described below.
- If you have no plan and no more intentions, then **do nothing**. This is what we named the **satisfaction state**. All the intentions of the agent are fulfilled. But notice, that the agent continues to process the messages sent to it. Thus it will respond to questions about its state and even may be reactivated by the instruction message 'move-away-from-me'.
- If you have no plan and the actual intention is fulfilled, then store the actual intention in the remember-list and **fetch the next intention** and make it the actual intention. At the very beginning in the first activation, the remembering is of course omitted.

➡ If all this does not apply, then **generate a new plan** from the actual intention and execute the first goal of this plan.

The following flow chart shows the procedure that is realized by these rules:



The conditions of the rules refer to **knowledge tokens** of the agent's private knowledge base, which realizes a kind of **internal status** of the agent. This status is changed during the processing of messages and the execution of actions to achieve goals. For example the knowledge token [act-intention-fulfilled] is asserted only during the execution of the goal 'test-intention' if the intended ON-relation is true in the actual world state.

The **five boxes** of the flow chart of the figure determine the agent's behavior within one activation:

The **processing of messages** has already been discussed in the previous section.

The **generation of a plan** is a very simple transformation of an intention into a plan, which is a sequence of simple plan steps. This has also already been discussed in the description of the planning layer.

The **satisfaction state** is a kind of sleeping state and a reactivation from that state is only the exception. The environmental loop of activations is terminated when all agents are in their satisfaction state.

If the actual intention is fulfilled, the **next intention** from the intention-list is going to be processed. This is also a very simple action performed by the predicate 'get_next_intention'.

The most important and totally by predicates of the action layer realized behavior is the **execution of plan steps**. The intended meaning of the different goals was already presented in the description of the planning layer. Now we sketch how **the predicate 'reach_goal'** achieves a goal.

The appended code once again shows the naturalness and elegance with which PROLOG expresses this procedures and shows - now after we have gained detailed knowledge about the other layers - the interdependencies of the action layer and the other layers. If the basic constructs of PROLOG are known and the terminology of the application is acquainted, comments to the PROLOG code are nearly superfluous. We present in each case the **goal**, a **description** of the actions to achieve it and the **code** that realizes these actions:

nil : The 'reach_goal' predicate is invoked with the empty goal nil when all plan steps have been worked out such that the current plan is the empty list. Therefore the knowledge token [i-have-a-plan] must be retracted.

```
%*** execution of plan steps:
reach_goal(Agent, nil) :-
        retract(iknow(Agent, [i-have-a-plan])).
```

know(X) : The existence of X must be asserted because X occurs in one of the agent's intentions that was used to generate the plan. If the position and the size are not already known, the agent must send a question and work on that goal again during the next activation when the answer have been received.

```
reach_goal(Agent, know(X)) :-
        (iknow(Agent, [existence, X]) ->
                true
        |
                assert(iknow(Agent, [existence, X]))),
        (iknow(Agent, [pos-size, X]) ->
                true
```

```
                    |
                        send_message(X,Agent,[question, where-are-you]),
                        insert_to_plan(Agent, know(X))).
```

test-intention : Replace the constant 'self' by the own name in the actual intention and use the ON -predicate of the spatial knowledge layer to test the truth value of the intended relation. If it holds, assert that the actual intention is fulfilled and that you have no plan anymore. Otherwise do nothing.

```
reach_goal(Agent, test-intention) :-
        act_intention(Agent, Intention),
        Intention =.. List,
        replace(List, self, Agent, Newlist),
        NInt =.. Newlist,
        (NInt ->    assert(iknow(Agent, [act-intention-fulfilled])),
                    retract(iknow(Agent, [i-have-a-plan]))
                |
                    true).
```

fix(X) : Inspect the private knowledge base for information about that. If none is available, then ask X after its move-intention, assert the knowledge token [asked-movement] to be aware of that question in the next activation, and try to achieve the goal fix(X) in the next activation again. If X is the constant 'self', then the asking behavior is replaced by a call of the introspective predicate 'intention_to_move'. At last, when the agent knows that X is fixed, the goal is achieved. Otherwise, if X intends to move, the plan is replaced by the single-goal plan [wait_for_move(X)]. (Confer on the next page how that goal is achieved if X is 'self'!)

```
reach_goal(Agent, fix(self)) :-
        iknow(Agent, [i-am-fixed]).
reach_goal(Agent, fix(self)) :-
        (intention_to_move(Agent) ->
                    assert(iknow(Agent, [i-intend-to-move])),
                    retract(plan(Agent, P)),
                    assert(plan(Agent, [wait_for_move(self)]))
                |
                    assert(iknow(Agent, [i-am-fixed]))).

reach_goal(Agent, fix(X)) :-
        iknow(Agent, [is-fixed, X]).
reach_goal(Agent, fix(X)) :-
        iknow(Agent, [intends-to-move, X]),
        retract(plan(Agent, P)),
        assert(plan(Agent, [wait_for_move(X)])).
reach_goal(Agent, fix(X)) :-
        send_message(X, Agent, [question, you-intend-to-move]),
        insert_to_plan(Agent, fix(X)),
        assert(iknow(Agent, [asked-movement, X])).
```

free(X) : In principle similar to the previous goal: First inspection of the private knowledge and appropriate actions, if information is available. If not, a question is

posted to X or broadcasted to everyone if X is 'self', and the goal is kept till the next activation and the awareness of the question is assured in the knowledge base and in the case of broadcasting additionally in the plan.

```
reach_goal(Agent, free(self)) :-
        iknow(Agent, [i-am-free]).
reach_goal(Agent, free(self)) :-
        iknow(Agent, [is-on-me, X]),
        send_message(X, Agent, [instruction, move-away-from-me]),
        retract(plan(Agent, P)),
        assert(plan(Agent, [wait_for_move(X)])).
reach_goal(Agent, free(self)) :-
        position(Agent, [X, Y]),
        size(Agent, [Len, _]),
        (broadcast_message(Agent,[question,someone-on-me,[X Y],Len]) ->
                true
        |
                insert_to_plan(Agent, free(self)),
                assert(iknow(Agent, [asked-freedom])),
                insert_to_plan(Agent, answer-freedom)).

reach_goal(Agent, free(X)) :-
        iknow(Agent, [is-free, X]).
reach_goal(Agent, free(X)) :-
        iknow(Agent, [not-free, X, is-under, Y]),
        retract(plan(Agent, P)),
        assert(plan(Agent, [wait_for_move(Y)])).
reach_goal(Agent, free(X)) :-
        send_message(X, Agent, [question, are-you-free]),
        (iknow(Agent, [asked-freedom-of, X]) ->
                true
        |
                assert(iknow(Agent, [asked-freedom-of, X]))),
        insert_to_plan(Agent, free(X)).
```

wait_for_move(X) : The waiting behavior is realized by focusing the agent's attention to the knowledge token [has-moved, X], that is asserted during the message processing. If that token occurs, the knowledge base is updated and the waiting ends. Otherwise the waiting goal is again inserted into the plan. A special case is if an agent waits for an own move. This kind of internal deadlock is broken by the predicate 'follow_another_intention', that fetches a new intention from the intention-list, makes it the actual intention and puts the old actual and still unfulfilled intention to the end of the intention-list.

```
reach_goal(Agent, wait_for_move(self)) :-
        retract(iknow(Agent, [i-have-a-plan])),
        follow_another_intention(Agent).

reach_goal(Agent, wait_for_move(X)) :-
        (iknow(Agent, [has-moved, X]) ->
                retract(iknow(Agent, [has-moved, X])),
```

```
                (iknow(Agent, [is-on-me, X]) ->
                        retract(iknow(Agent, [is-on-me, X]))
                    |
                        true)
            |
                insert_to_plan(Agent, wait_for_move(X))).
```

move_on(X) : First of all the movement is announced to all other agents by broadcasting. The rest of the procedure is a little bit technical and computes the exact start- and goal-coordinates for the movement. If X is 'ground', a subroutine 'next_free_ground' searches for a free space on the table. The movement itself is done by the subroutine 'agent-movement', which splits the movement into several little steps and calls a predicate of the world window, which generates commands for picture drawing after each step.

```
reach_goal(Agent, move_on(ground)) :-
        (broadcast_message(Agent, [announcement, i-jump-on, ground]) ->
                true
            |
                true),
        position(Agent, [Startx, Starty]),
        size(Agent, [W, H]),
        ground_level(Y),
        Goaly is Y - H,
        next_free_ground(Agent, Goalx),
        steps(Steps),
        Xsteps is (Goalx - Startx)/Steps,
        Ysteps is (Goaly - Starty)/Steps,
        absolute(Xsteps, XA),
        absolute(Ysteps, YA),
        agent_movement(Agent,[Startx,Starty],[Goalx,Goaly],
                                [Xsteps,Ysteps],[XA,YA]).

reach_goal(Agent, move_on(X)) :-
        (broadcast_message(Agent, [announcement, i-jump-on, X]) ->
                true
            |
                true),
        position(Agent, [Startx, Starty]),
        size(Agent, [W, H]),
        position(X, [Goalx, Y]),
        Goaly is Y - H,
        steps(Steps),
        Xsteps is (Goalx - Startx)/Steps,
        Ysteps is (Goaly - Starty)/Steps,
        absolute(Xsteps, XA),
        absolute(Ysteps, YA),
        agent_movement(Agent,[Startx,Starty],[Goalx,Goaly],
                                [Xsteps,Ysteps],[XA,YA]).
```

pause : Very simple: Just do nothing.

```
reach_goal(Agent, pause).
```

answer-freedom : If another agent has responded to the question [someone-on-me], the predicate 'process_message' would have updated the private knowledge base. Therefore, if that is not the case, (i.e. no [is-on-me, Y]-token is in the knowledge base,) the agent assumes to be free.

```
reach_goal(Agent, answer-freedom) :-
        iknow(Agent, [is-on-me, Y]).

reach_goal(Agent, answer-freedom) :-
        retract(iknow(Agent, [asked-freedom])),
        assert(iknow(Agent, [i-am-free])).
```

A few further predicates, that serve as routines and to make the code more readable, also belong to the action layer. They are too technical (e.g. the search for a free place on the table) or too simple and self explaining (e.g. 'insert_to_plan') to discuss them here in more detail. The following sections will go on with the description of the other modules of the system structure, which was presented in section 2.2.: The current world scenario and the environment.

5. The "World"

The current world scenario of the blocksworld program comprises three major functions: The **agent status module** shows all important activities and the internal state of the agents. The **blackboard** is split into one part that provides **communication ports** for each agent and one part, that presents a **world window** view to the user. To support a **pictorial presentation of the scene** to the user, a simple drawing procedure was written in C that uses the X Window system to open a window and draw the scene on it. As an interface to this **drawing process**, a simple set of commands was defined. The PROLOG part of the world window generates these commands and the drawing process interprets and executes them to draw the picture.

The **agent status module** accesses the agents' layered knowledge bases at different levels, presents the accessed information, which may include abstraction, condensation or transformation into other representational formats, and presents the result to the user. The information flows only in one direction, the agent status module has no authority to manipulate anything in the layered knowledge base. Therefore its task can be compared with that of a **filter for information**. The agent status module of our simple blocks world system is realized by the predicate showstatus, which uses the Quintus PROLOG built-in predicate 'listing' to flush informations about the agents intentions, plans, private knowledge, remember-lists and all exchanged interagent messages on the screen.

The idea of a **blackboard** in general is that of a high level shared memory. The agents have read- and write-access to that structure. Each agent can write knowledge pieces

(hypotheses) onto the blackboard and all other agents will “see” it. In a typical blackboard architecture, the board is the place, where the global solution is developed.

In our simple blocksworld simulation, the blackboard is a **communication device** for the agents and will inform the user about the current state of the world. The first is done by providing communication ports for the agents. The second is the task of the world window part of the blackboard.

The **communication ports** are realized with the predicate 'blackboard'. Sending a message means assertion of a fact of that predicate that has the form: blackboard(sender-agent, receiver-agent, message-token). An agent A can only access those messages, that specify A as the receiver agent. This is a simple and robust (no loss of messages and **arrival guarantee** within the next activation cycle !) way to implement a communication channel.

The other task of the blackboard assigned to the **world window** is to provide information about the actual state of the world. This can be done in a similar way as the agent status module by flushing information about the world state on the screen. A **pictorial presentation of the scene** would be much more comfortable. We decided to use the X Window system to implement a simple and robust drawing process. It was written in C and works as follows: First a window is opened and initialized for drawing. The process waits for commands from the input stream. Each command causes a drawing action which is visible on the window. This process runs independent from the PROLOG session and one problem is the connection between PROLOG and that drawing facility. As a logical interface we designed a **set of primitive drawing commands**, that are generated by the PROLOG-part of the world window and interpreted and executed by the drawing process. This will be described in more detail below. Our first idea was to couple PROLOG and the drawing process via a pipe: PROLOG will put the commands into the write side of the pipe, while the drawing process gets them out on the read side. The operating system will do the process synchronization. Unfortunately Quintus PROLOG, which we used for our implementation, did not support any facility to conduct the data of an output stream into a pipe. A first attempt to program a simple C procedure for that task and bind it into the PROLOG program failed because the facility to bind foreign language code into PROLOG, which was described in the manual, did not work. Therefore we used a file as a buffer: PROLOG writes the drawing commands on a file. The drawing process reads this file and calls the X-routines to draw the scene picture. The PROLOG part of the world window consists of procedures to open and close the file and a procedure to generate the commands. The later one calls a predicate to draw the table and another to draw the agents, which accesses the lowest layer of the agents' knowledge base to get the x,y-coordinates and sizes of them. That looks like this:

```
%********* (drawing functions):  **********************************************
initdraw :-
        open(proto, append, Stream),
        assert(drawstream(Stream)).
```

```
exitdraw :-
        retract(drawstream(Stream)),
        close(Stream).

drawpic:-
        drawworld,
        drawagents,
        flushpic.

flushpic :-
        drawstream(Stream),
        print(Stream,'('),
        print(Stream,(clear)),
        print(Stream,')'),
        nl(Stream).

drawworld :-
        drawstream(Stream),
        ground_level(Y),
        x_range([Min, Max]),
        print(Stream,'('),
        print(Stream,(fillrect,Min,Y,Max,20)),
        print(Stream,')').

drawagents :-
        drawstream(Stream),
        agentlist(AL),
        member(Agent, AL),
        position(Agent,[X,Y]),
        size(Agent,[W,H]),
        print(Stream,'('),
        print(Stream,(rect,X,Y,W,H)),
        print(Stream,')'),
        X1 is X + 5,
        Y1 is Y +20,
        print(Stream,'('),
        print(Stream,(string,X1,Y1,Agent,1)),
        print(Stream,')'),
        fail.
drawagents.
```

We already see from that how the graphic commands look like: Each **command** is enclosed in brackets and consists of a colon-separated sequence of strings. The first string identifies the command and the following strings are the parameters. The following list presents all commands, that the drawing process understands:

(point, x, y)	:	Draw a point at the coordinates (x,y).
(line, x1, y1, x2, y2)	:	Draw a line with starting point (x1,y1) and ending point (x2,y2).

(rect, x, y, w, h)	:	Draw a rectangle with upper left corner at point (x,y) and width w and height h.
(fillrect, x, y, w, h)	:	Draw a filled rectangle with upper left corner at point (x,y) and width w and height h.
(arc, x, y, w, h, a1, a2)	:	Draw an arc that fits into a rectangle with upper left corner at (x,y) with width w and height h. Parameter a1 specifies the start of the arc relative to the three-o'clock position from the center. Parameter a2 specifies the length of the arc specified in 64ths of a degree. (360*64 is a complete circle).
(fillarc, x, y, w, h, a1, a2)	:	Same as the previous command, but draws a filled segment of a circle from the starting point to the ending point of the specified arc.
(string, x, y, s, l)	:	Writes the string s at position (x,y). The length is specified by l. The X Window system allows to use different fonts.
(clear)	:	Copies the drawn picture to the window so that it gets visible to the user.

Not all of these commands are used in our system for our scenarios need no circles to be drawn. The last command is necessary because all drawings are first done to a pixmap, which is not visible to the user. The clear command copies this map to the window and frees the pixmap for the next picture. This is a good method to hide the process of drawing from the user and generate a picture without flickering.In more time-critical scenarios, several pixmaps could be used the produce a kind of animated cartoon.

6. The Environment

The basic module in our pictorial representation of the system's structure in section 2.2. is labeled as environment. The meaning of environment in the context of multi-agent systems in general is not uniform in different systems. The environment should represent the world or field where the agents exist. It is used to **group agents together within time and space** and make them react to changes of the environment.

In our blocksworld simulation, we count every part of the program to the environment, that does not realize specialities of the agents. That means all predicates, that have no agent-parameter. (Those predicates, that realize the agent status module and the world window part of the blackboard, are excepted.) The functionality of these predicates realize the top-level **activation-loop**, a very simple **time** concept, the **initialization** of the scenario, especially the **specification of the start- and goal-scene** by the user and the **translation** of the goal scene **into the agents local intentions**, which is a simple form of task decomposition.

The **activation-loop** repeats activating the agents one after the other using their activity predicate. A list of all agents is used for that and the sequence in this list determines the sequence of activation. The **termination condition** is, that all agents are satisfied, which is tested by inspecting the agents' private knowledge bases whether the knowledge token [no-more-intentions] is asserted. This is done by the predicate 'until-everyone-is-satisfied', which in case of failure causes PROLOG to backtrack to the predicate 'repeat'. This behavior of PROLOG implements the loop. A clipping of the code in shown in the following:

```
%*** main program and activation-loop:
main :-
        initialize,
        activationloop.

activationloop :-
        repeat,
        showstatus,
        activate_all_agents,
        drawpic,
        tick,
        agentlist(AL),
        until_everyone_is_satisfied(AL),
        exitdraw.

activate_all_agents :-
        agentlist(AL),
        member(Agent, AL),
        activity(Agent),
        fail.        % backtracking assures the activation of all members of the agentlist.
activate_all_agents.

tick :-
        retract(time(T)),
        T1 is T + 1,
        assert(time(T1)).

until_everyone_is_satisfied([]).
until_everyone_is_satisfied([Agent | Rest]) :-
        iknow(Agent, [no-more-intentions]),
        until_everyone_is_satisfied(Rest).
```

The simple **time** concept also gets obvious from that piece of code. It is not more than a predicate, that counts the number of times that the activation-loop turns around. This is done by 'tick' and the time turned out to be a useful information for debugging.

The **initialization** of the scenario comprises the specification of the **agents**, the **start-scene** and the **goal-scene** by the user. Both scenes are specified in a different way: The agents and their initial positions are specified within a **user dialogue** that proceeds like this:

system's question:	**user input**:
You want a new agent?	yes or no
What is its name?	name of the new agent
Position of the agent?	name of an already specified supporting agent or 'ground'
Size of agent?	'd' as short for default-size or 'size(W,H)', where W is the width and H the height of the agent.

The dialogue ends with a 'no' to the first question and the system's request to specify a goal-scene. This is done in form of a **list of ON-relations**, which is also the internal form of representation for the initial scene generated from the dialogue:

What should be the final scene?	list of ON-relations, e.g.: [on(a,b),on(b,c), ...]

A simple check controls if all agents of the goal-scene are specified in the previous dialogue, but this does not guarantee a **consistent goal scene**. An inconsistent scene description will lead to a **faulty behavior**. To make the system more fault tolerant could be a starting point for future work. The global scenes are specified in terms of ON-relations, therefore the **exact x,y-coordinates** of the blocks **must be computed** by a predicate 'computeinitscene' which calls several subpredicates to perform the technical details of that computation. A simple **task decomposition** is performed by the predicate 'getownintentions', which extracts each agent's intentions out of the global scene representation. For each agent, those ON-relations, that mention the agent, are collected and the agent-name is replaced by the constant 'self'.

Finally we want to describe how the blocksworld simulation system can be started: You have to start a Quintus PROLOG session and consult the file 'blockprog', which contains all the code discussed in the previous sections. To start a scenario you have to invoke the predicate 'main', which will start with the user dialogue to specify a scenario. To shortcut that dialogue it is possible to write all answers of the user in a file and start the scenario by the invocation of 'defaultmain', which will ask you only for the filename.

7. Conclusions

The presented system shows how in a simple toy-world multi-agent scenario the interaction of the agents can be organized to lead to a problem solving behavior where no global view of the problem is necessary.

The interest was focused on the *communication* and the *introspective knowledge* of the agents. Both are very tightly entangled as for example can be seen in the answering behavior: When an agent receives an answer like “yes” or “no”, it must associate its previously asked question to it, which in our simulation is done by using the agent's introspective knowledge. In general we used a communication based on several message types, where there exist several message tokens for each type. This concept could be developed further for more sophisticated scenarios (e.g. the cooperation of transportation

companies) by the introduction of more types or a hierarchy of types. It would also enlarge the flexibility of the communication in a multi-agent scenario if there would be methods to make an agent learn new message types or message tokens from other agents. But such a facility must also provide methods to modify the agents internal procedures to process messages on the one hand and to make the agents learn to use new introspective knowledge tokens on the other hand.

With exception of one relatively simple procedure, we organized the introspective knowledge of the agents in a similar way as knowledge tokens, i.e. as facts which are asserted or retracted dynamically during the run of a scenario. The set of all holding knowledge tokens at a timepoint can be interpreted as a kind of state of the agent. This state is then used to guide the agent's behavior, for example to decide about asking a question or not, to change its actual plan, or to generate a new plan out of its intentions.

The agent's planning behavior and its actions cannot be isolated from the higher level knowledge layers. The planning in our scenario comprises four levels: Intentions are resolved into plans, which consist of several goals. Each goal is achieved by simple actions. This may be a first step to a hierarchical planning behavior. A simple form of replanning is realized by the possibility, that an existing plan is modified e.g. as an effect of certain messages.

The concept of the agent-activation cycle seems to be appropriate for agent-societies of that level of complexity. It guarantees a certain flexibility, because what the agents do within their activation is completely defined by the agents and separated from the activation loop. Different scenarios with different types of agents can use that concept. To say it clear: The simulated concurrency depends on the “good will” of the agents: If an agent does not return the control to the environment it is never interrupted. Such a mechanism should be integrated into a more sophisticated future version of the system.

PROLOG is well suited as an implementation language for the layered knowledge base. Facts as well as skills of the agents are represented in a natural way. The drawback of less efficiency is made up for the advantage of clarity and well structuredness, which was crucial for our intention to have a simple and prototypical experimental simulation with a minimal expenditure of work and time.

8. References

BG88 Bond, A., Gasser, L.: Readings in Distributed AI, Morgan Kaufmann, Los Angeles, 1988

BMS91 Bürkert, H.-J., Müller, J., Schupeta, A.: RATMAN and its Relation to other Multi-Agent Testbeds, RR-91-09, Research Report of the German Research Center of Artificial Intelligence, 1991

Cha87 Chapman, D : Planning for Conjunctive Goals, in Artificial Intelligence 32, 333-377, 1987

DM91 Demazeau,Y., Muller, J.-P.: Decentralized Artificial Intelligence, Proc of the second workshop on Modelling Autonomous Agents in a Multi-Agent World, Elsevier Sc. Pub/North Holland, 1991

Fer90 Ferber, J.: The Framework of Eco-Problem Solving, in (DM91)

FN71 R.E.Fikes and N.J.Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2, 1971

Sch92 Schupeta, A.: Topics of DAI: A Review, RR-92-06, Research Report of the German Research Center of Artificial Intelligence, 1991

Wal75 Waldinger, R.: Achieving several goals simultaneously, SRI AI-Center, Tech. Note 107, Menlo Park, CA, 1975

Deutsches
Forschungszentru
m
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
67608
Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse oder per anonymem ftp von ftp.dfki.uni-kl.de (131.246.241.100) unter pub/Publications bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Research Reports

RR-93-14
Joachim Niehren, Andreas Podelski, Ralf Treinen: Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support Final Project Report
33 pages

RR-93-16
Gert Smolka, Martin Henz, Jörg Würtz: Object-Oriented Concurrent Constraint Programming in Oz
17 pages

RR-93-17
Rolf Backofen:
Regular Path Expressions in Feature Logic
37 pages

RR-93-18
Klaus Schild: Terminological Cycles and the Propositional μ-Calculus
32 pages

RR-93-20
Franz Baader, Bernhard Hollunder:
Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-22
Manfred Meyer, Jörg Müller:
Weak Looking-Ahead and its Application in Computer-Aided Process Planning
17 pages

DFKI Publications

The following DFKI publications or the list of all published papers so far are obtainable from the above address or via anonymous ftp from ftp.dfki.uni-kl.de (131.246.241.100) under pub/Publications.
The reports are distributed free of charge except if otherwise indicated.

RR-93-23
Andreas Dengel, Ottmar Lutzy:
Comparative Study of Connectionist Simulators
20 pages

RR-93-24
Rainer Hoch, Andreas Dengel:
Document Highlighting — Message Classification in Printed Business Letters
17 pages

RR-93-25
Klaus Fischer, Norbert Kuhn: A DAI Approach to Modeling the Transportation Domain
93 pages

RR-93-26
Jörg P. Müller, Markus Pischel: The Agent Architecture InteRRaP: Concept and Application
99 pages

RR-93-27
Hans-Ulrich Krieger:
Derivation Without Lexical Rules
33 pages

RR-93-28
Hans-Ulrich Krieger, John Nerbonne, Hannes Pirker: Feature-Based Allomorphy
8 pages

RR-93-29
Armin Laux: Representing Belief in Multi-Agent Worlds viaTerminological Logics
35 pages

RR-93-30
Stephen P. Spackman, Elizabeth A. Hinkelman:
Corporate Agents
14 pages

RR-93-31
Elizabeth A. Hinkelman, Stephen P. Spackman: Abductive Speech Act Recognition, Corporate Agents and the COSMA System
34 pages

RR-93-32
David R. Traum, Elizabeth A. Hinkelman: Conversation Acts in Task-Oriented Spoken Dialogue
28 pages

RR-93-33
Bernhard Nebel, Jana Koehler: Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis
33 pages

RR-93-34
Wolfgang Wahlster: Verbmobil Translation of Face-To-Face Dialogs
10 pages

RR-93-35
Harold Boley, François Bry, Ulrich Geske (Eds.): Neuere Entwicklungen der deklarativen KI-Programmierung — *Proceedings*
150 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

RR-93-36
Michael M. Richter, Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Gabriele Schmidt: Von IDA bis IMCOD: Expertensysteme im CIM-Umfeld
13 Seiten

RR-93-38
Stephan Baumann: Document Recognition of Printed Scores and Transformation into MIDI
24 pages

RR-93-40
Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Werner Nutt, Andrea Schaerf: Queries, Rules and Definitions as Epistemic Statements in Concept Languages
23 pages

RR-93-41
Winfried H. Graf: LAYLAB: A Constraint-Based Layout Manager for Multimedia Presentations
9 pages

RR-93-42
Hubert Comon, Ralf Treinen: The First-Order Theory of Lexicographic Path Orderings is Undecidable
9 pages

RR-93-43
M. Bauer, G. Paul: Logic-based Plan Recognition for Intelligent Help Systems
15 pages

RR-93-44
Martin Buchheit, Manfred A. Jeusfeld, Werner Nutt, Martin Staudt: Subsumption between Queries to Object-Oriented Databases
36 pages

RR-93-45
Rainer Hoch: On Virtual Partitioning of Large Dictionaries for Contextual Post-Processing to Improve Character Recognition
21 pages

RR-93-46
Philipp Hanschke: A Declarative Integration of Terminological, Constraint-based, Data-driven, and Goal-directed Reasoning
81 pages

RR-93-48
Franz Baader, Martin Buchheit, Bernhard Hollunder: Cardinality Restrictions on Concepts
20 pages

RR-94-01
Elisabeth André, Thomas Rist: Multimedia Presentations: The Support of Passive and Active Viewing
15 pages

RR-94-02
Elisabeth André, Thomas Rist: Von Textgeneratoren zu Intellimedia-Präsentationssystemen
22 Seiten

RR-94-03
Gert Smolka: A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards
34 pages

RR-94-05
Franz Schmalhofer, J.Stuart Aitken, Lyle E. Bourne jr.: Beyond the Knowledge Level: Descriptions of Rational Behavior for Sharing and Reuse
81 pages

RR-94-06
Dietmar Dengler: An Adaptive Deductive Planning System
17 pages

RR-94-07
Harold Boley: Finite Domains and Exclusions as First-Class Citizens
25 pages

RR-94-08
Otto Kühn, Björn Höfling: Conserving Corporate Knowledge for Crankshaft Design
17 pages

RR-94-10
Knut Hinkelmann, Helge Hintze:
Computing Cost Estimates for Proof Strategies
22 pages

RR-94-11
Knut Hinkelmann: A Consequence Finding Approach for Feature Recognition in CAPP
18 pages

RR-94-12
Hubert Comon, Ralf Treinen:
Ordering Constraints on Trees
34 pages

RR-94-13
Jana Koehler: Planning from Second Principles —A Logic-based Approach
49 pages

RR-94-14
Harold Boley, Ulrich Buhrmann, Christof Kremer:
Towards a Sharable Knowledge Base on Recyclable Plastics
14 pages

RR-94-15
Winfried H. Graf, Stefan Neurohr: Using Graphical Style and Visibility Constraints for a Meaningful Layout in Visual Programming Interfaces
20 pages

RR-94-16
Gert Smolka: A Foundation for Higher-order Concurrent Constraint Programming
26 pages

RR-94-17
Georg Struth:
Philosophical Logics—A Survey and a Bibliography
58 pages

RR-94-18
Rolf Backofen, Ralf Treinen:
How to Win a Game with Features
18 pages

RR-94-20
Christian Schulte, Gert Smolka, Jörg Würtz:
Encapsulated Search and Constraint Programming in Oz
21 pages

RR-94-31
Otto Kühn, Volker Becker, Georg Lohse, Philipp Neumann:
Integrated Knowledge Utilization and Evolution for the Conservation of Corporate Know-How
17 pages

RR-94-33
Franz Baader, Armin Laux:
Terminological Logics with Modal Operators
29 pages

DFKI Technical Memos

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben:
The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

TM-93-01
Otto Kühn, Andreas Birk: Reconstructive Integrated Explanation of Lathe Production Plans
20 pages

TM-93-02
Pierre Sablayrolles, Achim Schupeta:
Conlfict Resolving Negotiation for COoperative Schedule Management
21 pages

TM-93-03
Harold Boley, Ulrich Buhrmann, Christof Kremer:
Konzeption einer deklarativen Wissensbasis über recyclingrelevante Materialien
11 pages

TM-93-04
Hans-Günther Hein:
Propagation Techniques in WAM-based Architectures — The FIDO-III Approach
105 pages

TM-93-05
Michael Sintek: Indexing PROLOG Procedures into DAGs by Heuristic Classification
64 pages

TM-94-01
Rainer Bleisinger, Klaus-Peter Gores:
Text Skimming as a Part in Paper Document Understanding
14 pages

TM-94-02
Rainer Bleisinger, Berthold Kröll:
Representation of Non-Convex Time Intervals and Propagation of Non-Convex Relations
11 pages

DFKI Documents

D-93-15
Robert Laux:
Untersuchung maschineller Lernverfahren und heuristischer Methoden im Hinblick auf deren Kombination zur Unterstützung eines Chart-Parsers
86 Seiten

D-93-16
Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Gabriele Schmidt: Design & KI
74 Seiten

D-93-20
Bernhard Herbig:
Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus
97 Seiten

D-93-21
Dennis Drollinger:
Intelligentes Backtracking in Inferenzsystemen am Beispiel Terminologischer Logiken
53 Seiten

D-93-22
Andreas Abecker:
Implementierung graphischer Benutzungsoberflächen mit Tcl/Tk und Common Lisp
44 Seiten

D-93-24
Brigitte Krenn, Martin Volk:
DiTo-Datenbank: Datendokumentation zu Funktionsverbgefügen und Relativsätzen
66 Seiten

D-93-25
Hans-Jürgen Bürckert, Werner Nutt (Eds.):
Modeling Epistemic Propositions
118 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-26
Frank Peters: Unterstützung des Experten bei der Formalisierung von Textwissen
INFOCOM:
Eine interaktive Formalisierungskomponente
58 Seiten

D-93-27
Rolf Backofen, Hans-Ulrich Krieger, Stephen P. Spackman, Hans Uszkoreit (Eds.):
Report of theEAGLES Workshop on Implemented Formalisms at DFKI, Saarbrücken
110 pages

D-94-01
Josua Boon (Ed.):
DFKI-Publications: The First Four Years
1990 - 1993
75 pages

D-94-02
Markus Steffens: Wissenserhebung und Analyse zum Entwicklungsprozeß eines Druckbehälters aus Faserverbundstoff
90 pages

D-94-03
Franz Schmalhofer: Maschinelles Lernen:
Eine kognitionswissenschaftliche Betrachtung
54 pages

D-94-04
Franz Schmalhofer, Ludger van Elst:
Entwicklung von Expertensystemen:
Prototypen, Tiefenmodellierung und kooperative Wissensevolution
22 pages

D-94-06
Ulrich Buhrmann:
Erstellung einer deklarativen Wissensbasis über recyclingrelevante Materialien
117 pages

D-94-07
Claudia Wenzel, Rainer Hoch:
Eine Übersicht über Information Retrieval (IR) und NLP-Verfahren zur Klassifikation von Texten
25 Seiten

D-94-08
Harald Feibel: IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung
58 Seiten

D-94-09
DFKI Wissenschaftlich-Technischer Jahresbericht 1993
145 Seiten

D-94-10
F. Baader, M. Lenzerini, W. Nutt, P. F. Patel-Schneider (Eds.): Working Notes of the 1994 International Workshop on Description Logics
118 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-94-11
F. Baader, M. Buchheit,
M. A. Jeusfeld, W. Nutt (Eds.):
Working Notes of the KI'94 Workshop:
KRDB'94 - Reasoning about Structured Objects: Knowledge Representation Meets Databases
65 Seiten

D-94-12
Arthur Sehn, Serge Autexier (Hrsg.):
Proceedings des Studentenprogramms der 18. Deutschen Jahrestagung für Künstliche Intelligenz KI-94
69 Seiten