Operationalizing Conceptual Models Based on a Model of Dependencies

Frank Maurer* **& Jürgen Paulokat***

Abstract. In this paper we describe a framework for defining and operationalizing conceptual models of distributed knowledge-based systems which extends published approaches by the notion of „agents“ and multiple task decompositions. The main part deals with techniques underlying our distributed interpreter. We show how a client-server-architecture can be implemented which allows prototyping distributed knowledge-based systems. Further we describe our mechanism which manages task interactions and supports dependency-directed backtracking efficiently.

1 INTRODUCTION AND OVERVIEW

Model-based development of knowledge-based systems is the state of the art in knowledge engineering. A well-known approach in Europe is KADS [14] which allows to model knowledge-based systems on a very high level of abstraction. For this methodology, several formal and/or operational languages were developed [5]. These do not support the distribution of tasks among several agents. Our goal is the development of a framework which supports the description and prototypical operationalization of distributed knowledge-based applications. The framework was developed with the following requirements in mind:

- Modelling of distributed knowledge-based systems
  A description of the cooperation between several agents must be part of the conceptual model. Therefore, the framework supports the notion of „agents“ which in our opinion is essential for modelling distributed knowledge-based systems.
- Integration of human users in problem solving process
  In practical development projects often it is not possible to formalize every inference step so that it can done by computers: Some inferences should be drawn by human users who can use all their cognitive skills. Only the results must be given to the knowledge-based system to be used in the ongoing problem solving process.
- Selection of task decomposition at run-time
  Our framework is developed especially for modelling design tasks. Design problems normally can be solved in different ways. Only the current problem defines which is the appropriate way. Therefore, the model must allow several task decompositions and the appropriate model must be selected at run-time.[1]
- Dependency-directed backtracking must be supported
  For design tasks several task decompositions are useful. The appropriate decomposition is selected at runtime. Normally, the selection is based on additional assumptions which may lead to inconsistencies in the inference process. Dependency-directed backtracking, which improves the efficiency of the needed backtracking process, is not supported by currently implemented interpreters for conceptual models.

In chapter 2 we give a short overview on our terminology for modelling distributed knowledge-based systems. The interpreter for the models is described in chapter 3. This chapter is the core of the paper. In the last chapter, we summarize and discuss our approach.

2 MODELLING OF DISTRIBUTED KNOWLEDGE-BASED SYSTEMS

To describe distributed knowledge-based systems our methodology[2] uses four basic notions:[3] task, method, concept and agent. In the remainder of this chapter we will roughly describe them. A detailed description is given in [6]. In this paper, we focus on operationalization aspects.

Tasks and Methods: A task is specified by: What should be reached (the goal), what is used (the input) and what should be the result (the output). We distinguish between atomic and complex (composed) tasks: Atomic tasks[4] are solved by an agent by applying an atomic method. For complex tasks a set of complex methods can be defined, which can be used alternatively for solving the problem. Every complex method is described as a dataflow diagram, which consists of variables and (sub-)tasks. For every variable its type is defined, i.e. the concept class which should be instantiated at runtime is associated to the variable. Every subtask can be further decomposed by methods. In this sense, the decomposition of the overall task is described as an AND-OR-Tree, where tasks are the OR-Nodes and methods are the AND-Nodes. The decomposition which is useful for the problem at hand is selected at runtime, e.g. when the model is executed.

* University of Kaiserslautern, AG Expertensysteme / Prof. Richter, P.O. Box 3049, D-67653 Kaiserslautern, Germany, e-Mail: {maurer, paulokat}@informatik.uni-kl.de

ECAI 94. *11th European Conference on Artificial Intelligence* Edited by A. Cohn
Published in 1994 by John Wiley & Sons, Ltd.

1 As stated by one of the reviewers, selecting task decompositions at runtime exemplifies what is intended by the strategy layer of the KADS methodology.

2 Our methodology is supported by a computer aided knowledge engineering tool (CoMo-Kit) which is not described in this paper.

3 The notions are used similar to KADS and the Components-of-Expertise-Approach [10]. Therefore, we do not describe the syntactical details of our language for conceptual modelling.

4 Atomic tasks are our analogue to inference steps in KADS. The difference is that a atomic task can be decomposed when the knowledge engineering process is continued. In our approach there is no predefined finest granularity for inference steps.

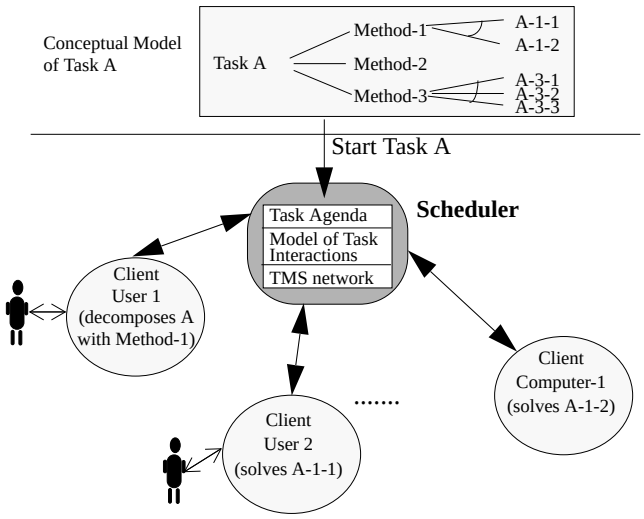

Figure 1: Architecture of the interpreter

Complex tasks are solved by performing an associated method. The solution of a complex task is supervised by an agent which is responsible that every subtask is done.

Concepts: To model data structures we follow an object-centered approach. We distinguish concept classes and concept instances. Classes specify the structure of instances by defining a set of attributes (slots). Every attribute is associated with a type which can be a concept class or a basic type (Symbol, String, Real, etc.).[5]

Agents: Tasks are solved by agents. In the conceptual model for every task a set of agents is defined. Every agent of this set has the ability to handle the task. We distinguish two kinds of agents: humans and computers. Depending on the kind of agents which are associated to it, a task must be described in natural language or in a operational language.

With the defined framework an abstract specification of a distributed knowledge-based system - a conceptual model - may be described. The next question is how to operationalize this specification, i.e. how to implement an interpreter for these specifications.

3 OPERATIONALIZATION OF CONCEPTUAL MODELS

In the first section, we describe the architecture of our interpreter. In section 3.2 we discuss how our interpreter manages the task agenda. The last section deals with our mechanisms which manages dependencies between tasks and supports backtracking efficiently.

3.1 Architecture of the Interpreter

For the operationalization of conceptual models we use techniques known from AI planning. Our interpreter contains a dependency management mechanism which supports dependency directed backtracking.

To solve tasks, methods are applied. Complex methods decompose tasks into subtasks and define variables. Primitive or atomic methods assert values to variables.

The interpreter has a Client-Server-Architecture. The server is a scheduler which stores pending tasks and (valid) variable bindings. For managing task interactions it uses an extended TMS (cf. section 3.3). Tasks are performed by clients: A client accesses a task, selects an appropriate method, applies it, and transfers the results[6] back to the server. The communication between server and client extends the protocol defined in [6].

Figure 1 explains the mode of operation of the interpreter: There exists a conceptual model of task A with 3 methods. Task A is started. User-1 selects method-1 to decompose task A and reports this to the scheduler. Because of this, tasks A-1-1 and A-1-2 enter the tasks-to-be-performed-agenda. After that, user-2 accepts task A-1-1 and computer-1 solves task A-1-2. The result is shown in Figure 1. In the next section we describe the task management in more detail.

3.2 Task Management

In this section we describe the mode of operation of the interpreter with the state transition diagram for tasks (cf. figure 2). It shows possible states of one task or subtask during problem solving. for every state the scheduler maintains a list which contains all tasks currently belonging to this state.

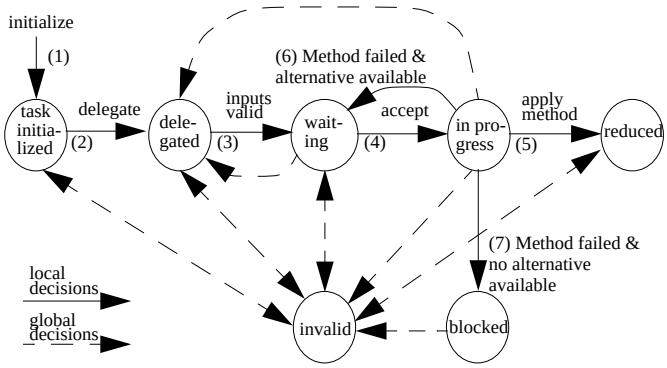

Figure 2: State transitions of tasks

The state transitions of tasks are managed by the scheduler. Tasks are started by committing them to the scheduler (1). As a result they are initialized.

After the initialization of a task the scheduler needs to know which agents shall work on the task. This information is passed to the scheduler when the task is delegated to the responsible agents (2). The delegation depends on the current situation: some agents may be ill or on vacation.

When all inputs are available the task changes its state to „waiting“ and waits to be performed (3). The inputs become available when all tasks which are in front of the waiting task in the dataflow diagram are finished and produced their output.

To accept a task the client logs in, asks the scheduler wether there are waiting tasks, and (finally) accepts the task (4). Then the task is marked as „in progress“.

The client solves the task: A method is applied. If the application is successful the task is reduced (5). If it fails and there are other

5 Based on class specifications we automatically generate graphical user interfaces (GUIs) for editing instances. The user interfaces can be further refined by an graphical user interface builder.

6 Results are subtasks or variable bindings.

methods available, the task re-enters the state „waiting“ (6). If the method fails and there are no alternative methods available, the task is blocked (7).

In step (5), (6) or (7), dependency structures for efficient backtracking are created (cf. section 3.3).

Up to now, all state transitions were performed because of decisions which are local for a task. The other transitions visible in figure 2 (with dashed lines) take place after global decisions (Meta Decisions). For example, a task becomes invalid only if it the scheduler globally decides that the problem may be solved without solving the task. This happens when an alternative decomposition for a task is selected during backtracking. Then subtasks which were introduced by the first method become invalid.

The need for backtracking can be explained with the Blocked-State as an example: A task is blocked if there is no method applicable which can be used to reduce the task. But the task reduction is necessary to solve the overall problem. This means that the problem solving process is in an inconsistent state. The inconsistency can only be solved by backtracking which takes the global state of the inference process into consideration: Some previously made decisions must be retracted and an alternative method must be selected for the corresponding task. The computation of a consistent state after the retraction of some decisions is supported by the dependency management mechanisms described in the next section.

3.3 Dependencies between Tasks

Solving a complex task results in numerous subtasks and applied methods. If a task can be reduced by alternative methods, it defines a decision point in the space of all solutions where one alternative must be chosen. Often, when a decision has to be made, complete information whether an alternative belongs to a consistent solution or about its impact on a solution’s quality is unavailable such that a problem solver must search, testing more or less systematically alternative choices until a consistent and satisfying solution has been found. This process can be made more efficient by search heuristics or by user interactions preferring the most promising alternative, but, inconsistent choices can not always be excluded such that a problem solver must be able to backtrack. A simple method to solve inconsistencies is chronological backtracking that, however, is extremely inefficient and tedious for a user.

3.3.1 REDUX - An architecture for decision revision

This poor behavior can often be improved by dependency-directed backtracking, but it needs a model of interactions between decisions. To support dependency-directed backtracking in our architecture, we extended REDUX [7], [8] that provides a general model of planning and design, emphasizing decision revisions and propagating their effects. The contract of REDUX is representing the search space for a problem solver and to provide it with current information about the state of tasks, methods, and decisions.[7] In this framework a task can be *unreduced*, *reduced* or *blocked*. It is unreduced until a method of its conflict set that explicitly or implicitly represents the set of all applicable methods, has been selected. The basic operation provided by REDUX for a problem solver is choosing an unreduced task and to reduce it by selecting a method with the consequence that the former unreduced task becomes reduced. For every selection, REDUX creates a decision that stores contingencies and rationales for its optimality. A contingency is an unexpected future event whose appearance immediately invalidates the selection with the consequence that the task becomes unreduced, again. The loss of validity of the rationales for a choice's optimality is less rigid. It does not automatically invalidate the decision, but creates an *optimality loss* meta task.[8] This task forces the problem solver to make a (meta) decision whether to improve the solution by changing the selected alternative or to go on with a suboptimal selection.

During problem solving, the growing set of valid decisions becomes inconsistent if selected methods result in constraint violations such that backtracking becomes necessary. Then, at least one decision must be retracted. The conditions that make a retraction necessary are represented by a rejection rationale whose validity retracts the decision and the method selected by it. Discriminating between the necessity of a retraction and the retraction itself allows a decision being retracted in situations different from that described by rejection retionals. If all methods of a conflict set have a valid rejection rationale, then the task is blocked because it can not be reduced. Then, the problem solver has to make a meta decision for solving such a *task block*. This can be done by retracting the method in whose dataflow diagram the blocked task is contained as a subtask or by retracting a method that caused the rejection of, at least, one element of the conflict set. These strategies are described more exhaustive in [8].

3.3.2 Dataflow dependencies

In the basic version of REDUX, complex methods only model the decomposition of tasks into subtasks without any orderings or dataflow dependencies. For this reason we extended the definition of a complex task by variables and dataflow relations between variables and subtasks. The variables of a method definition can be declared as input, output, or local. Input and output variables represent the interface of a method and are comparable to formal parameters of conventional programming languages. When applying a method, they are substituted by the variables of the input and output relations of the task being reduced, whereas for every local variable a unique instance is generated. In the dataflow defined by a method, input variables can only be used in the input relations of its subtasks, local and output variables can be used in input and output relations, and every output variable must be used in, at least, one output relation. In figure 3 the relations between tasks resulting from the application of complex method m_1 to task t and the resulting mapping of the method's input and output variables on the variables in the task's input and output relation are shown.

Primitive methods represent formal or informal[9] procedures for assigning values to variables and must not create new subtasks. While reducing a task by a primitive method, its procedure is executed resulting in assignments to the variables of the reduced task’s output relations. Assignments are used to represent data computed by method execution. In figure 4, the execution of atomic method m_{11} must result in assignments to variables l_1 and l_2. For its compu-

7 In the REDUX ontology the terms *goal* and *operator* are used that are replaced by *task* and *method* in this paper, respectively.

8 The notion of „meta tasks“ comes from planning and should not be confused with the notion of „tasks“ in knowledge engineering.

9 Informal procedures represent a textual description of actions intended to be executed by a human.

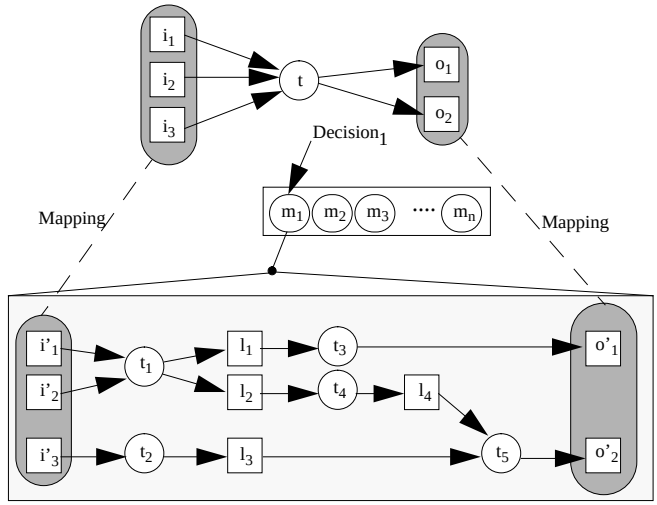

Figure 3: Complex method application: m_1 is applied to t and reduced to the dataflow at the bottom

tations it can use data represented by the current assignments of the variables i'_1 and i'_2. Because of earlier backtracking some variables have more than one assignment, but at most one of them is valid that is called the current assignment. It is important to note that an assignment will never be deleted. But, it can become invalid because of the retraction of the method that created it. However, as soon as this method becomes valid again, its assignments become valid, too, such that the reevaluation of its procedure is not necessary.

If the same data are computed by different methods, they are represented by the same assignment. These methods are called the supporters of the assignment that becomes valid as soon as one of its supporters is valid. Representing identical data with the same assignment, minimizes the number of performing procedures. E.g., if atomic method m_{41} of the conflict set of tasks t_4 in figure 3 uses as input the assignment a_{22}, it becomes valid as soon as method m_{12} or m_{13} is chosen because both are supporting the used assignment. So, the assignments resulting from applying method m_{41} must be computed only once although there are at least two different situations in which it can be applied. Unique representation of computed data and storing assignments that have become invalid, as described above, is especially important if the computation is expansive or done by a human because solving the same task twice is not acceptable.

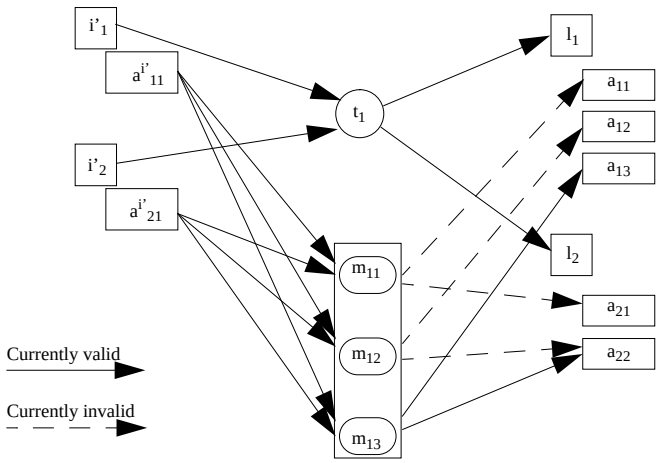

Figure 4: Primitive method application

3.3.3 TMS-based dependency management

Based on these extended dependencies between tasks as well as between tasks and variables, we can model the states of the diagram in figure 2. An unreduced task is not executable as long as the variables in its input relation are unassigned. The variables in the output relation of an unreduced, but executable task become assigned, if an atomic method has been applied to it. After retracting a decision, all variables being assigned only by the method selected by this decision, again become unassigned. If a variable of a task's input relation becomes unassigned, the method reducing it will be retracted. So, a local change can ripple through large parts of a dataflow and the task hierarchy resulting in a lot of tasks changing their state resulting in a new consistent set of valid tasks, decisions, variables, and assignments.

In REDUX a TMS [3] is used to represent and efficiently propagate dependencies and to support dependency-directed backtracking if inconsistencies manifest. For every choice a TMS structure is generated, build up from TMS nodes and justifications, as shown in figure 5. Depending on such a network a choice is valid if the decision node is labeled IN. A node is labeled IN if it has a valid justification. Justifications are valid if all nodes in the inlist are labeled IN and all nodes in the outlist are labeled OUT. The elements of an inlist are represented by solid lines and those of the outlist by dashed lines. A node is labeled OUT if it does not have any valid justification. Labeling is automatically done by the TMS. The networks of all choices are connected by their valid task and valid subtask nodes and build up an overall dependency structure representing the search space and the dependencies between decisions.

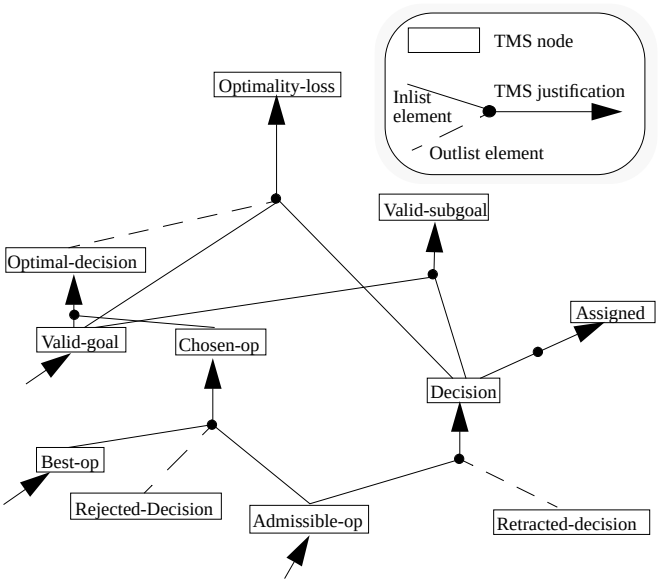

Figure 5: Standard decision dependencies (from [8])

The extended TMS structure supporting the dataflow between tasks is shown in figure 6. The *decision* and *rejected decision* nodes are part of the standard decision dependencies of figure 5 that are only partially shown is this figure. Applying a complex method additionally creates an *unassigned* and an *assigned* node for every variable of its definition and an *executable* node for every subtask. Because of their justifications, unassigned nodes are initially labeled IN and the assigned nodes are labeled OUT. The executable node for a task becomes IN as soon as the unassigned nodes for the variables in the task's input relation are labeled OUT. This signals the problem

solver that an unreduced task is executable. If it executes an atomic method, the decision node of the generated standard decision dependencies supports the assignment nodes of the computed data. For every assignment node a new justification for the according assigned node is generated, such that the assigned node is labeled IN, if there is an assignment with an IN label. However, if an assignment node for a datum already exists, then only a new justification for this node is created.

Based on the indirection using the unassigned node in the outlist of the justification for the executable node instead of using the assigned node in its inlist, the executability of a subtask depends on the validity of the decision that selects the method in whose dataflow the subtask is contained. This is necessary because all retracted decisions and their effects are stored for later reuse. But problem solving is focused on currently valid tasks by only selection task that are executable.

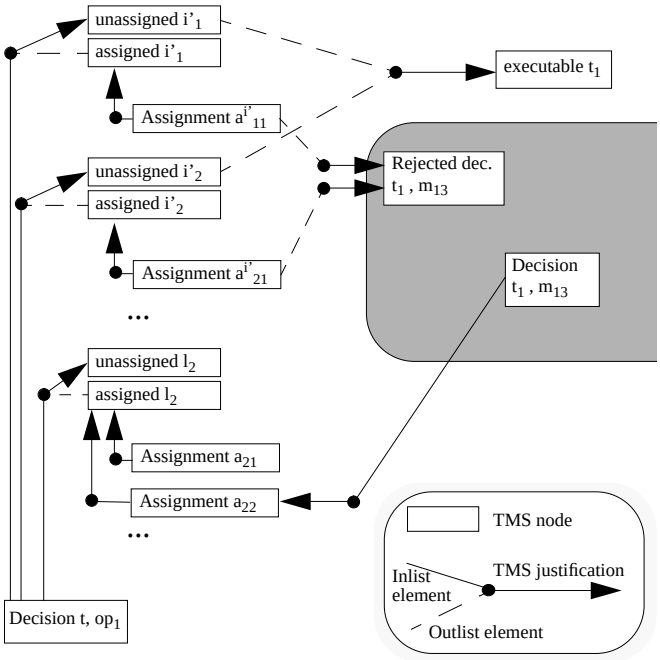

Figure 6: Exemplary dataflow dependencies

4 SUMMARY AND DISCUSSION

In this paper we described a framework for modelling distributed knowledge-based systems. Extensions of current work are the notion of agents and multiple methods for task decomposition. Further, we explained how an interpreter for this extended conceptual model works. Especially, we discussed how dependencies between tasks are managed so that backtracking is efficiently supported.

The interpreter is build based on a client-server-architecture and supports distributed problem solving. Tasks may be executed by human or computer agents, i.e.methods may be applied not only by computers. Using our approach one is not forced to formalize every inference step before the overall task may be solved by a system prototype.

Our approach deals with the four problems mentioned in chapter 1 and is an extension of several approaches for the operationalization of conceptual models.

Modelling of distributed knowledge-based systems is supported by a fully implemented knowledge engineering tool (CoMo-Kit). Currently, we extend our interpreter which is a part of this tool by the dependency management mechanisms described in section 3.3. Further, we started a project which uses the techniques described in this paper for a complex real-world domain (Zoning).

ACKNOWLEDGMENTS

We would like to thank Barbara Dellen and Willi-Gerd Schmitz for discussing and implementing the ideas described in this paper.

REFERENCES

[1] Angele, J.; Fensel, D.; Landes, D.; Studer, R: KARL: An Executable Language for the Conceptual Model. In: Proceedings of the Knowledge Acquisition for Knowledge-Based Systems, Workshop KAW'91, October 6-11, Banff, 1991.

[2] Breuker, J.; Wielinga, B.; van Someren, M.; de Hoog, R.; Schreiber, G.; de Greef, P.; Bredeweg, B.; Wielemaker, J.; and Billault, J.-P.: Model-Driven Knowledge Acquisition: Interpretation Models. Esprit Project P1098, University of Amsterdam (The Netherlands), 1987.

[3] Doyle, J.: A Truth Maintenance System, Artificial Intelligence, 12:231-272, 1979.

[4] de Greef, H. P., Breuker, J. A.: Analyzing System-User Cooperation in KADS, In [9].

[5] Fensel, D., van Harmelen, F.: A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, Research Report, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, No. 280, September 1993.

[6] Maurer, F.: Hypermediabasiertes Knowledge Engineering für verteilte wissensbasierte Systeme, Dissertation University of Kaiserslautern, 1993, also: DISKI 48, infix-Verlag, ISBN 3-929037-48-3 (in german).

[7] Petrie, Ch.: Context Maintenance, in: Proceedings of AAAI-91, Menlo Park, California, 1991, MIT Press.

[8] Petrie, Ch.: Planning and Replanning with Reason Maintenance, Dissertation, University of Texas, Austin, 1991.

[9] Schreiber, G. (Ed.): Special Issue: The KADS Approach to Knowledge Engineering, Knowledge Acquisition, Vol. 4 No. 1, March 1992, Academic Press.

[10] Steels, L.: Components of Expertise, AI Magazine, 11 (2 (Summer)), 1990.

[11] van Harmelen, F., Balder, J.: $(ML)^2$: A Formal Language for KADS Models of Expertise, 1992, in [9].

[12] Wetter, Th.: First-order Logic Foundation of the KADS Conceptual Model, 1990, in [13].

[13] Wielenga, B., Boose, J., Gaines, B., Schreiber, G., van Someren, M. (ed.): Current Trends in Knowledge Acquisition, IOS Press, Amsterdam, May 1990.

[14] Wielinga, B.J.; Schreiber, A.Th.; Breuker, J.A.: KADS: A Modelling Approach to Knowledge Engineering. ESPRIT Project P5248 KADS-II, An Advanced and Comprehensive Methodology for Integrated KBS Development, Amsterdam, 1991.

[15] Wielinga, B.J.; Schreiber, A.Th.; Breuker, J.A.: KADS: A Modelling Approach to Knowledge Engineering, 1992, in [9].