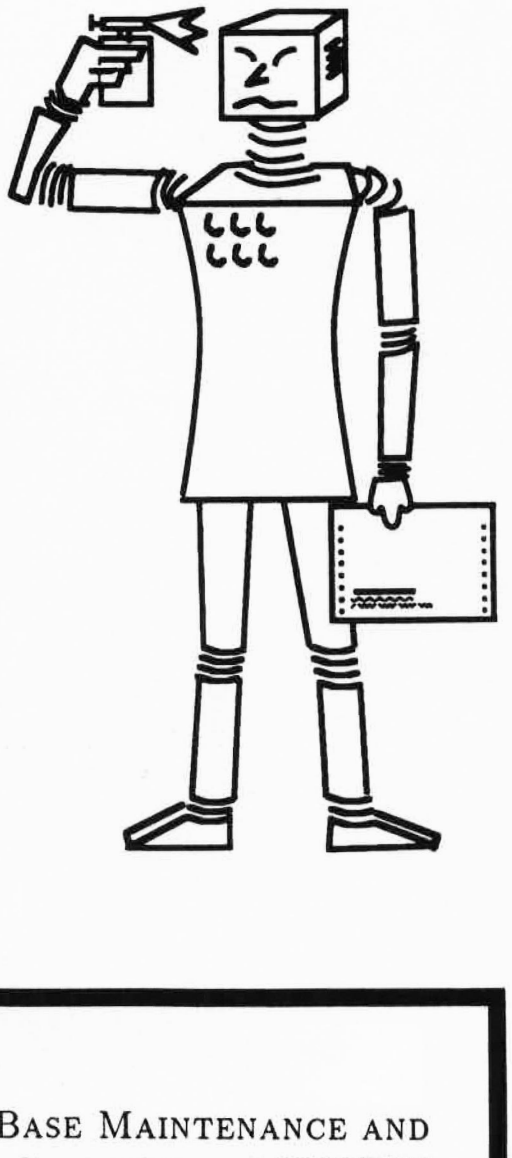
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT



Knowledge Base Maintenance and Consistency Checking in MOLTKE

Frank Maurer

SEKI Report SR-91-10 (SFB)

Knowledge Base Maintenance and Consistency Checking in MOLTKE[1]

Frank Maurer[2]
University of Kaiserslautern
Dept. of Computer Science
P.O. Box 3049
D-6750 Kaiserslautern
Germany
e-Mail: maurer@informatik.uni-kl.de

ABSTRACT

This paper deals with special problems of knowledge base maintenance which have to be solved within the knowledge acquisition process. We illustrate that aspects of maintenance must be taken into account by the design model construction because dependencies between pieces of knowledge can result in inconsistent states of a knowledge base. We describe a *Knowledge Dependency Network* which extends ideas from truth maintenance systems to detect and manage these inconsistencies. The network allows formal definitions of inconsistency conditions and checks them automatically preserving the integrity of the knowledge base. As a fundamental part of the acquisition and maintenance environment the knowledge dependency network supports the conventional development and editing of a knowledge base.

KEYWORDS

knowledge acquisition, knowledge maintenance, consistency checking, expert system

[1] also: Proc. of the 6th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Canada, 1991

[2] The work presented herein was partially supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314: "KI - Wissensbasierte Systeme", projects X6 and X9.

1. OVERVIEW

This paper deals with aspects of knowledge acquisition, knowledge base maintenance and consistency checking. Our experience in expert system projects showed that aspects of knowledge maintenance must be considered while constructing the design model of the domain. After defining appropriate knowledge structures, there are often troubles with inconsistencies in the knowledge base because of dependencies between knowledge objects. Often only the co-existence of a few objects results in an inconsistent state.

Our general approach to knowledge acquisition and maintenance is described in chapter 2. We illustrate the approach by following the development of the MOLTKE shell for technical expert systems. In paragraph 2.1 we introduce the basic representation language for which an interpreter is presented in 2.2. Then we present our general approach to knowledge maintenance (2.3). We define the knowledge dependency network (2.4) and the concept of graph consistency (2.5). The network enables the knowledge engineer to specify conditions of consistency for knowledge bases. It manages the dependencies of knowledge objects and automatically maintains the consistency of a knowledge base. The network is an extension and a new application of the ideas of truth maintenance systems ((de Kleer, 86), (Doyle, 79)). The resulting maintenance component supports the user while inserting and deleting objects preserving the consistency of the knowledge base. Paragraph 2.6 gives some complexity estimations. Chapter 3 of this paper presents an overview on the other parts of the MOLTKE workbench for technical diagnosis. The workbench is a complex, fully implemented expert system toolbox which integrates several second generation expert system techniques. In chapter 4 we discuss results and compare our approach with related work. Chapter 5 gives an overview on our ongoing work and the state of realization.

2. SOME REQUIREMENTS OF KNOWLEDGE ACQUISITION AND MAINTENANCE

In general the building of experts systems consists of two main parts. First a model of the domain has to be constructed. The expert(s) and the knowledge engineer usually work together in this phase. This model is implemented on a computer by the knowledge engineer who usually has a background in computer science and artificial intelligence. The second part includes the filling of the model with the expert´s knowledge. Ideally, the domain expert himself fills the knowledge base because:

- only he can guarantee for the correctness of the system,

- he is able to test whether the system acts appropriately,

- he often has to maintain the knowledge base (in fact: he is the only person who can decide whether the represented knowledge has to be changed).

A problem is that experts usually do not have any background knowledge concerning expert systems. Therefore, the representation formalism must reflect the experts own terminology. This observation was our reason to develop the design model for technical diagnosis described in 2.1 and 2.2.

To summarize the above mentioned one can say that if the representation formalism uses the terminology of the domain, the understanding of the knowledge base and the process of inference is simplified for the expert. Therefore, maintenance is made easier. At least the testing and debugging of the knowledge base may better be supported by the expert then.

Furthermore, a development tool has to support the editing and maintenance of the knowledge base, especially if used by a domain expert who is usually not used to work with computer systems. A knowledge maintenance component has to include facilities which easily allow to create and test knowledge bases. It has to satisfy the following requirements:

- incremental input of the knowledge in an user-determined order (this is the natural way of editing a knowledge base for an expert),
- managing the changes which follow an insertion or deletion of an object,
- showing dependencies of the represented pieces of knowledge,
- showing the effects of changes in the knowledge base (often the overview is lost if much knowledge is represented),
- checking the syntactic correctness and the semantic consistency (see 2.4) of the knowledge base.

Watching the knowledge acquisition process we find several steps which possibly must be repeated. At first we construct a conceptual and a design model of the domain (see also KADS (Breuker, Wielinga, 87)) which collect all relevant concepts of the domain (within the MOLTKE project for diagnosis we developed the representation language described below)[3]. The model does not include all facts of the domain. It is only a frame which will be filled in a later step. But it is immediately refined to the implementation level[4].

Extending the KADS framework we now ask for dependencies between the relevant aspects of the domain and for inconsistency conditions (Chapter 4 includes a discussion of the use of a truth maintenance system for managing the dependencies). The next step is to implement a domain-dependent acquisition interface which is used to fill the knowledge base. Thereby the consistency of the entered knowledge is automatically preserved by the knowledge dependency network described below.

We now illustrate the knowledge acquisition process described above by tracing the development of the MOLTKE shell for technical diagnosis. Thereby we concentrate on the knowledge base maintenance component.

[3] These steps will be supported by our hypermedia-based knowledge engineering environment HyperCAKE which is briefly described in chapter 5.

[4] So our view combines a KADS-like approach with a rapid-prototyping one.

2.1. Basic Definitions

In the MOLTKE project we developed a design model for technical diagnosis[5]. We believe that diagnosis can be described as follows:

<u>Diagnosis = Classification + Test Selection</u>.

A knowledge base represents the knowledge about the technical system divided into these two parts. Now we describe the basic vocabulary:

A *symptom class* relates a name to a list of possible values, its *type*, (e.g.Valve --> {open, closed}) whereas *symptom instances* reflect the actual state of a part of the technical device (e.g. Valve 21Y5 --> open). The actual value may be either *unknown* or an element of the list of possible values in the corresponding class.

The set of all symptom instances is called a *situation*. Within the context of predicate calculus the actual situation is the base for the interpretation of a *language of formulas*. It stores the variable bindings[6]. For the evaluation of formulas we use a three-valued logic with *true, false* and *unknown*.

A *test* ascertains the value of one or more symptom instances[7]. Usually, a test asks the user for the value of the symptom instance. The sequence of testing is determined by a set of ordering rules where the left hand side is a formula and the right part contains the symptom instance to test (e.g. (if (= Valve 21Y5 open) then relais test)).

To express relations between symptom values *shortcut rules* are used (e.g. (if (= light room-1 on) then wires := working)).

Contexts are one of the means for modularization in MOLTKE. A context represents a rough, intermediate, or final diagnosis. If its precondition is true, the associated failure is said to be proven and the related *correction* is executed[8]. Any context contains a set of ordering rules which locally prescribe the strategy of testing. Additionally, a context includes a set of shortcut rules. A correction describes what has to be done when a special fault occurs. For example, a context (without rules) is defined by the following statement:

```
Context name: LIGHT-BULB
        precondition: (SWITCH = CLOSED) & (LIGHT = OUT)
        correction: "Change the light-bulb"
```

The contexts are organized in a *context graph.* Its arcs have the semantics "is-refinement-of" (e.g. the context *failure-in-electric* is a refinement of *failure-in-car*.).

5 The design model was developed by the mechanical engineers of the WZL (machine tool laboratory) from the RWTH Aachen and our knowledge engineers.

6 Every symptom instance is a variable in the calculus.

7 In the formal sense of modal logic a test describes the transition from one world to another where the interpretation of the formulas differs in the bindings of the examined variables.

8 Normally only a text is printed on the screen.

2.2. The Interpreter

A global interpreter, which is easy to adapt and to maintain because it is organized in small modules, processes the knowledge base. The interpreter uses an establish-and-refine strategy.

The diagnostic process goes through the context graph by testing symptoms according to the ordering rules of the actual context and switching to a refinement when its precondition becomes (the logical value) true. If a leaf of the context graph is reached the system prints the diagnosis and terminates.

2.3. Knowledge Maintenance in MOLTKE

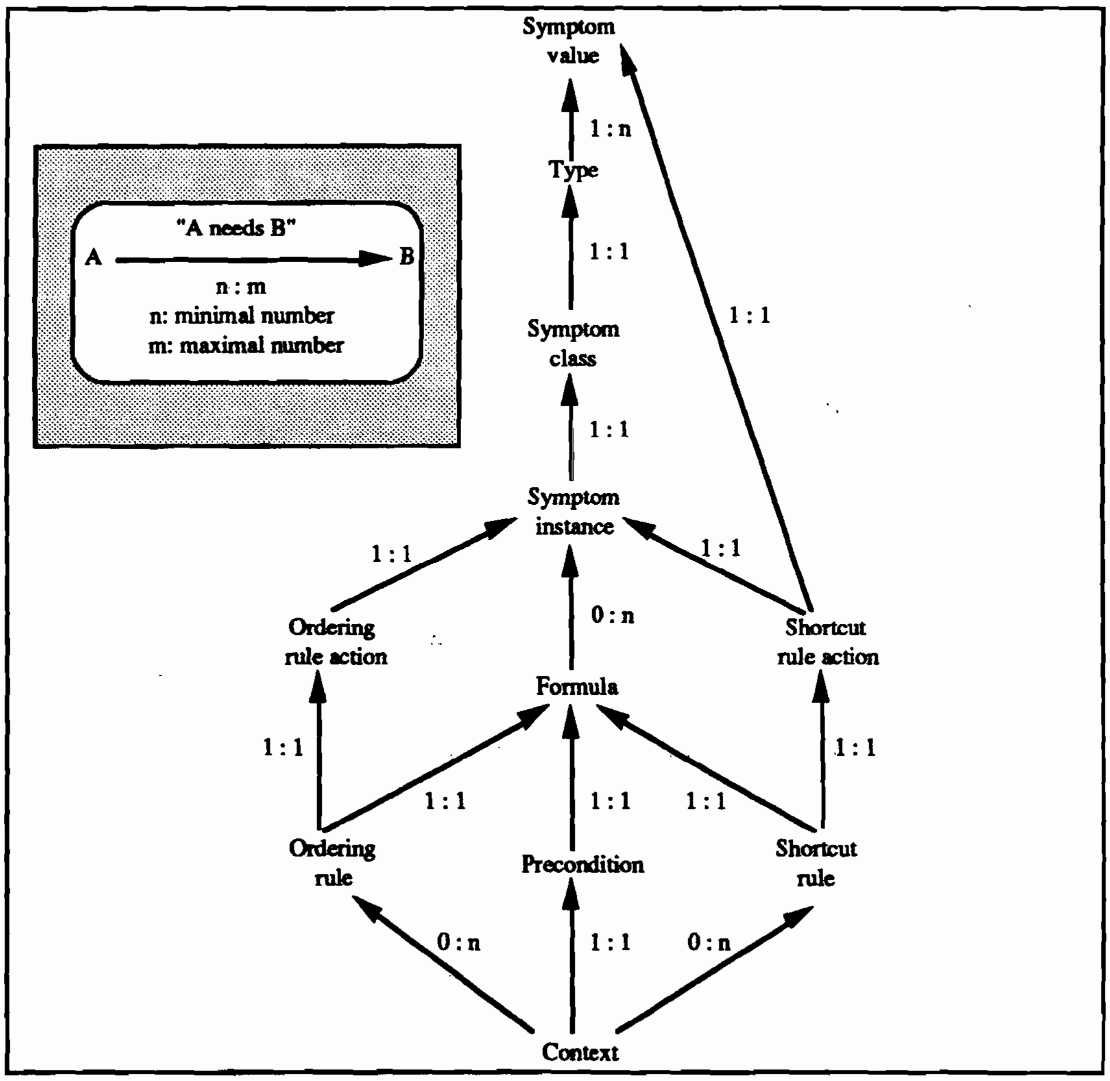


Figure 2.1: Dependencies of MOLTKE objects

After developing the basic representation language and implementing the appropriate parsers for the domain-dependent acquisition interface we asked our engineers for the dependencies of the defined structures. The answers are shown in figure 2.1.

Extending the acquisition interface which uses the parsers for checking the syntactic correctness of a knowledge base we built the MOLTKE maintenance component which checks two further aspects of consistency:

- node consistency: a piece of knowledge is used in the inference process only if all referred objects are correctly defined
- graph consistency: the knowledge base must not contain redundancies and contradictions

In the following paragraphs we illustrate these concepts of consistency.

2.4. Knowledge Dependency Network

To ensure the integrity of a knowledge base we build up a knowledge dependency network, which supervises dependencies between knowledge chunks.

For each kind of knowledge objects (e.g. in MOLTKE types, rules, contexts, etc.) a node class is defined which is used as a pattern for the definition of instances for the knowledge base. Figure 2.2 shows two examples for node class definitions.

```
(1)   Symptom-Instance-Node
            needs: (Symptom-Class-Node(1:1))
            needed-by: ()
            is-a-source: ()
(2)   Context-Node
            needs: (Precondition-Node(1:1)
                    Shortcut-Rule-Nodes(0:n)
                    Ordering-Rule-Nodes(0:n)....)
            needed-by: ()
            is-a-source: ()
```

Figure 2.2: Node class definitions of the knowledge dependency network for MOLTKE objects

Instances of these classes are used to represent all knowledge objects defined by the user. But only "correct" objects (e. g. objects which are consistent with the rest) will be inserted into the knowledge base. E. g. example (1) above states that a symptom instance node is only correctly defined if the needed symptom class was inserted into the knowledge base before.

The instances are the nodes of the knowledge dependency network. For each node exists an unambigous identifier, usually the name of the corresponding knowledge object. Arcs represent three different relations: needs, needed-by and is-a-source. The needs-relation expresses the necessary preconditions for a correct object. The creator of a node is referred to by the is-a-source-relation. The needed-by-relation shows where a node is used.

Each node instance stores three lists (needs-, needed-by- and is-a-source-list) which contain the anchors of the above described relations[9]. These lists are filled by the insertion of new nodes into the network (see below).

The knowledge dependency network separates the knowledge base from the developer of the expert system (see figure 2.3). It stores all syntactically correct object definitions. But objects are only inserted into the knowledge base if their definition is consistent with the rest of the knowledge base. Only objects in the knowledge base are used in the inference process. In fact the network supervises the non-monotonic process of filling a knowledge base[10]. Additionally, an agenda stores objects which are needed but not defined. Entries in the agenda contain the name of the needed object, its type and the node where it is referred.

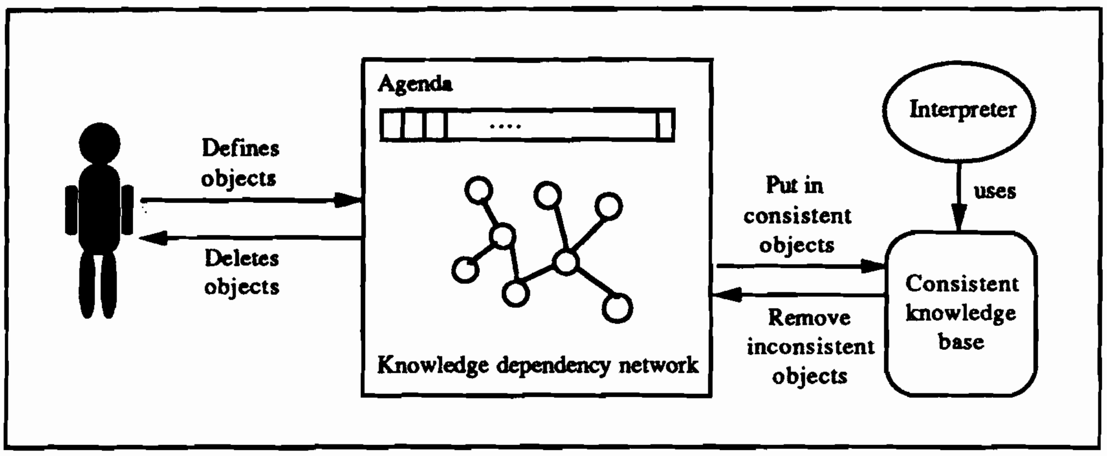

Agenda
....
Defines objects
Deletes objects
Knowledge dependency network
Put in consistent objects
Remove inconsistent objects
Interpreter
uses
Consistent knowledge base

Figure 2.3: Separating the expert from the knowledge base

The approach of knowledge dependency networks may always be used in domain-dependent models because we also found the defined relations when we talked with our experts in construction and planning domains. Also, the algorithms for the creation and deletion of new knowledge objects are general and not domain specific. They do not use references to the special kind of objects used in diagnosis.

2.4.1. Defining new Knowledge Objects

If the user wants to create a new object for the knowledge base the system first checks the syntax of the definition. Then all entries of the needs-list must be filled with an object corresponding to the type used in the node class (e.g. a symptom-instance-node needs a symptom-class-node as the corresponding anchor).

If all referred objects are already included in the knowledge base, the new object will also be put in. Otherwise the user is informed and the needed anchors are put on the agenda of objects which must be defined in the future. In this case the new object is not put into the knowledge base (preserving its consistency). This feature allows the

[9] For the relation *rel(A,B)* we define object *B* as the anchor of object *A* and the other way around.

[10] Inserting or deleting an object may result in an inconsistent state of the knowledge base. Conclusions drawn from a prior state may then be faulty. In fact, dependencies of objects may be viewed also as inferences (e.g. from the existence of one object the system may infere that another object has to exist too).

user to define new pieces of knowledge in whatever order he wants and nevertheless only consistent knowledge bases are used for inference. If all nodes are completely anchored we call the network *node consistent*.

A node consistent state may also be reached by a given order of definitions, which means that an object may only be defined if all needed objects are already existing. The knowledge dependency network supports the construction of a knowledge base because the developer is not unnecessarily restricted.

Then the is-a-source-list is filled with the origin of the object. Usually, the source is the expert who edits the knowledge base. The is-a-source-list is used if an object shall be deleted. Only the creator of an object may remove it.

At the end of an object's insertion into the knowledge dependency network the system checks if it is already needed in another node (which means that a reference is included in the agenda). If so, the needs- and needed-by-lists of the two objects are updated. Possibly, the other object is now node consistent. Then it will be put into the knowledge base and its needed-by-list is checked for other nodes which may change into a consistent state. Figure 2.4 shows the algorithm for inserting a new node into the network[11].

```
1.      Parse the definition of the new object generating a list of all referenced node
        instances
1.1.      Parsing ok: Mark the new node as consistent
1.2.      else: reject the new definition
2.      For all needs-relations of the class corresponding to the new node instance do
2.1.      Check if the referenced object is already in the network
2.1.1.      If true: Create the needs- and needed-by-relations between the two
            nodes;
2.1.2.      If false: Enter the referenced node into the agenda and mark the new
            node as inconsistent;
3.      if the new node is consistent then put it into the knowledge base;
4.      Store the source(s);
5       For all references to the new node in the agenda do
5.1.      Create the needs- and needed-by-relations between the two nodes;
5.2.      if the object from the agenda is consistent then put it into the knowledge
          base (then all dependent objects must also be checked for consistency and are
          possibly entered into knowledge base);
```

Figure 2.4: The algorithm for inserting of a new node into the knowledge dependency network

2.4.2. Deleting Knowledge Objects

While the need-list stores the essential preconditions of an object and is filled by insertion, checking the pattern of the node class, the dependent objects are included in the needed-by-list which helps the developer to delete objects.

If an object is removed by the user, the system checks if it is referred to somewhere else. If so, the user is informed about this fact and may decide if he really wants to

[11] See Appendix A for a detailed example.

delete the object. As a result of a consequent deletion, the dependent objects are marked as inconsistent and removed from the knowledge base. Figure 2.5 shows the algorithm for deleting a node.

```
1.    For all objects stored in the needs-list do
1.2.     Remove the needs- and needed-by-relation between the two objects;
2..   If the node was consistent then remove it from the knowledge base;
3     For all objects o stored in the needed-by-list do
3.1.     Remove the needs- and needed-by-relation between the two objects;
3.2.     Generate an entry in the agenda that the deleted node is referenced;
3.3.     "object o is inconsistent"
         remove o from the knowledge base (then all dependent objects also must be
         marked and removed)
```

Figure 2.5: The algorithm for deleting a node from the knowledge dependency network

2.5. Graph Consistency of a Knowledge Base

A node consistent network includes locally correct nodes, which means that all their preconditions are fulfilled. Additionally, the integrity of a knowledge base must also be guaranteed for relations between more than two objects. The system must prevent that the combination of several objects results in an inconsistent state where the concept "Inconsistency" may only be defined according to the domain. A MOLTKE knowledge base for diagnosis must not contain for example

1. types with different names and same values,
2. symptom classes with the same type,
3. shortcut rules with the same precondition and different values for the same symptom instance on the right hand sides,
4. more than one context with the same precondition.

Based on the knowledge dependency network these conditions may be checked by a matching of subgraphs. This fits into our perspective of knowledge engineering which is strongly influenced by object-oriented analysis and object-oriented design. Therefore our approach is a little bit different from more logic-based validation techniques.

Figure 2.6 shows the structure of inconsistent subgraphs corresponding to the above conditions. Our default assumption is that all graphs which do not contain inconsistency subgraphs represent consistent states of a knowledge base. A knowledge dependency network which does not contain *inconsistency graphs* is called *graph consistent*.

While developing a design model of the domain we also acquire knowledge about inconsistencies and define the appropriate inconsistency subgraphs for the network. When a new node enters the network, the system, after having checked for node consistency, tests via graph matching if the network stays graph consistent. Otherwise the user is informed and has to resolve the inconsistency.

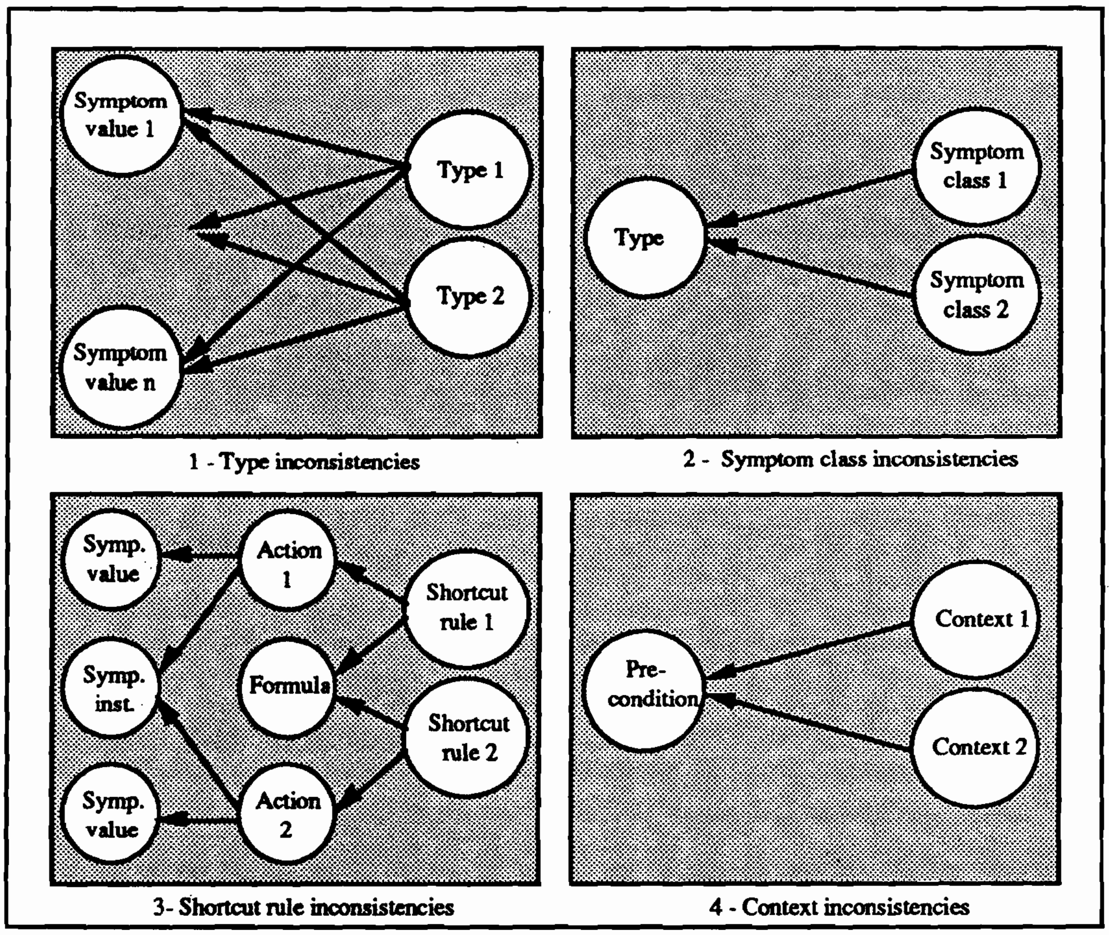


Figure 2.6: Inconsistent subgraphs of a knowledge dependency network for MOLTKE KBs

2.6. Complexity Estimations

The time needed for node insertion depends on the number *n* of entries into the needs-list, the size of the agenda *a* and the number of objects *o* which are dependent from the new object. So inserting a new node needs at most $O(n*a+o*a) = O(a*(n+o))$ steps. The algorithm for deleting a node needs $O(n+o)$ steps.

In our applications these estimations result in small numbers. E. g. in our CNC machining center domain (see (Richter, 92)) both, n and o, are smaller than 10. Checking graph consistency seems more problematic because it is reduced to graph matching. It is known that graph matching in general takes an exponential number of steps. In MOLTKE knowledge bases this effort is in fact drastically reduced because we first take advantage of having typed nodes. Therefore not all nodes need to be used as a starting point for a check. Secondly, the subgraphs are not very complex.

For the inconsistency graphs described above we have the following upper limits:

1. types with different names and same values:

*O(|number of types| * |max. size of types|)*

2. symptom classes with the same type:

   O(|number of needed-by-relations of the type|), because when inserting a symptom class the system needs to check only for the referenced type if it is used in another symptom class

3. shortcut rules with the same precondition and different values for the same symptom instance on right hand sides

   O(|number of needed-by-relations of the precondition|), because when inserting a shortcut rule the system needs to check only if it is used in another shortcut rule with a different value for the same symptom instance on right hand side

4. different contexts with the same precondition

   O(|number of needed-by-relations of the precondition|), because when inserting a context the system needs to check only if it is used in another context.

All in all we have a tolerable[12] time complexity for consistency checking of MOLTKE knowledge bases.

3. THE MOLTKE WORKBENCH FOR TECHNICAL DIAGNOSIS

The work reported here is only a small part of the MOLTKE project. Within this project we developed the above described conceptual model (in the sense of KADS (Breuker, Wielinga, 87)) for diagnostic expert systems in technical domains. This model was refined to the implementation level via a design model. Then we supplemented the model with qualitative reasoning and machine learning techniques. A detailed description of the resulting MOLTKE workbench is beyond the scope of this paper. Therefore we only give a brief overview and refer to already published papers.

In addition to the facilities described, in this paper we use second generation expert system technologies to support the knowledge acquisition process ((Althoff, Maurer, Rehbold, 90)). These include a model compiler which generates the core of a knowledge base out of a deep model of the technical system ((Rehbold 89), (Rehbold, 91)). This basic expert system is improved and extended by machine learning methods. We implemented a case-based reasoning system which supports the diagnostic process ((Althoff, Maurer, Weß, 91)). (Althoff, Maurer, Traphöner, Weß, 90) and (Althoff, 91) describe a system which learns relations between symptom patterns, and the work of (Maurer, Ruppel, 90) covers a system which learns diagnostic strategies based on neural network methods. The representation of temporal knowledge in the MOLTKE system is subject of ((Nökel, 89) (Nökel, 91)).

[12] Tolerable means that an interactive development of the knowledge base is possible.

4. DISCUSSION AND RELATED WORK

The MOLTKE workbench was developed in cooperation with the WZL, a mechanical engineering institute of the technical university of Aachen, following a pragmatic approach. Because of the integration of several advanced expert system techniques, the workbench contributes to the state of art in knowledge acquisition (following the requirements of (van Someren, Zheng, Post, 90)).

In this paper we described the maintenance component which supports the elicitation and coding of the knowledge by a knowledge dependency network. The network is especially helpful if

- serveral persons work together to build the knowledge base,
- the knowledge base is filled directly by the domain expert.

The network fulfills the above stated requirements on a maintenance component. Based on the network we are developing an interface which shows the effects of changes in the knowledge base. The network shows and supervises the dependencies of the pieces of knowledge and allows the user to define objects in an arbitrary order. Additionally, it checks the represented knowledge for inconsistencies. Therefore, it supports the construction of a knowledge base, especially if the builder of the domain model is different from the one who has to fill it.

Seeking in knowledge acquisition literature we find many contributions which are concerned with the more or less automatic acquisition of the domain models itself (e.g. (Morik, 86), (Breuker, Wielinga, 87)). We agree with the KADS group that a methodology for knowledge acquisition exists. We extend their approach by emphasizing aspects of maintenance which are very important if the expert system is supposed to live for a longer period of time.

The MOLTKE workbench can be used at least for every technical system of a similar complexity as a CNC machining center, which was our first application[13]. So we followed a more generic approach than (Musen, Fagan, Combs, Shortcliff, 86).

Our approach based on the knowledge dependency network is to fill and maintain a given domain model preserving formally defined consistency conditions. Scanning knowledge acquisition literature we did not find many papers on knowledge dependency supervising and consistency checking. Comparing our maintenance component with the approach of (Jansen, Compton, 89) we find two main differences. First, we use dependencies between pieces of knowledge for consistency checking; they mainly build cross references (which is only the first step of our approach). Secondly, they want to integrate different general representation formalisms (e.g. production rules, semantic nets, frames, etc.) whereas we want to ease the development of a knowledge base within a domain-dependent representation.

Supervising dependencies and backtracking to consistent states is the subject of truth

[13] Additionaly, we implemented expert systems for fault diagnosis in heterogenous computer networks, driving machines in mining, and CNC measuring machines.

maintenance systems (ATMS (de Kleer, 86), TMS (Doyle, 79)). Usually truth maintenance systems are used within one (dynamic) inference process. We need it for preserving a consistent state of a (static) knowledge base. So our system would be at least a new application of TMS.

Although we took a lot of inspirations from TMS approaches, we had to extend them. We have to deal with consistent and inconsistent states of the knowledge base. A currently inconsistent state may be consistent in the next step and then again become inconsistent (e.g the user defines an object, removes it and then redefines it). This swapping between consistency and inconsistency is not well handled by truth maintenance systems. Furthermore, a TMS does not find the inconsistencies. It is externally told that the actual state is inconsistent and then backs up to a consistent state. We need a system where we can define what is inconsistent and which then checks this by itself. Therefore we developed the knowledge dependency network.

A problem in the discussion of the MOLTKE workbench is that it consists of several complex components which deal with different topics out of the field of expert systems (e.g. deep modelling, qualitative reasoning, representation, acquisition, machine learning, knowledge maintenance and compilation). The integration of the components is an advantage over any stand-alone solution[14]. But this advantage can not be presented successfully within a paper dealing with special aspects of the whole system.

5. STATE OF REALIZATION AND ONGOING WORK

The MOLTKE base system for diagnosis (i.e. the above described representation formalism) is fully implemented in Smalltalk-80. Its graphic-oriented acquisition interface uses different parsers for checking the syntactic correctness of an object's definition. The deep modelling and machine learning facilities are implemented too.

Based on our experience we are developing a hypermedia-based knowledge engineering environment (called CAKE or HyperCAKE) which supports the knowledge acquisition process (Maurer, 91). The maintenance network will be integrated in this environment to support multiple experts and multiple knowledge engineers working together to built an expert system. It will support the process of model construction by allowing a smooth transition from informal (data level) descriptions via a semi-formal conceptual model to formal design models.

The HyperCAKE system uses the hypertext abstract machine (HAM) for storing and retrieving the informations (following the ideas of (Campbell, Goodman, 88)). We extend the HAM by facilities for typing nodes, links and contexts. Additionally, we integrate a rule interpreter into the hypertext machine.

We finished the implementation of the HAM and integrated it with a rule interpreter. The implementation of the maintenance component (which includes the described knowledge dependency network and consistency checkers) will be finished this year.

We are re-implementing the MOLTKE shell based on the HyperCAKE system until the beginning of 1992.

[14] in the sense of: "The whole is more than the sum of its parts".

6. ACKNOWLEDGEMENTS

I would like to thank the research group of Prof. Richter for the excellent working climate. Especially, I want to thank Prof. Richter for a lot of advices to this paper and Klaus Althoff, Alvaro de la Ossa, Hans Lamberti, Erica Melis, Scarlet Nökel, Jürgen Paulokat, Reinhard Praeger, Robert Rehbold, Mike Stadler and Holger Wache for discussing preliminary versions of this paper written in a really ugly English. Last but not least I would like to say a big "thank you" to Claudia Dell for the time she spent in maintaining me when I was writing this paper on knowledge base maintenance.

APPENDIX A

In the following we give an example how the editing of a knowledge base is supported by a knowledge dependency network. First, we define the node classes. For checking graph consistency the inconsistent subgraphs of figure 2.6 are used. Then, we define a few objects and show how the knowledge dependency network developes. The description of the needed parsers is left out.

Node class definitions:

Type-Node needs: ()
Symptom-Class-Node needs: (Type-Node (1:1))
Symptom-Instance-Node needs: (Symptom-Class-Node (1:1))

Defining objects for the knowledge base:

Step 1: Symptom-Instance new_with_name: SWITCH-1
for_class: SWITCH

What happens? A symptom instance node is put in the empty knowledge dependency network and marked as inconsistent. An entry for the needed symptom class is put into the agenda.

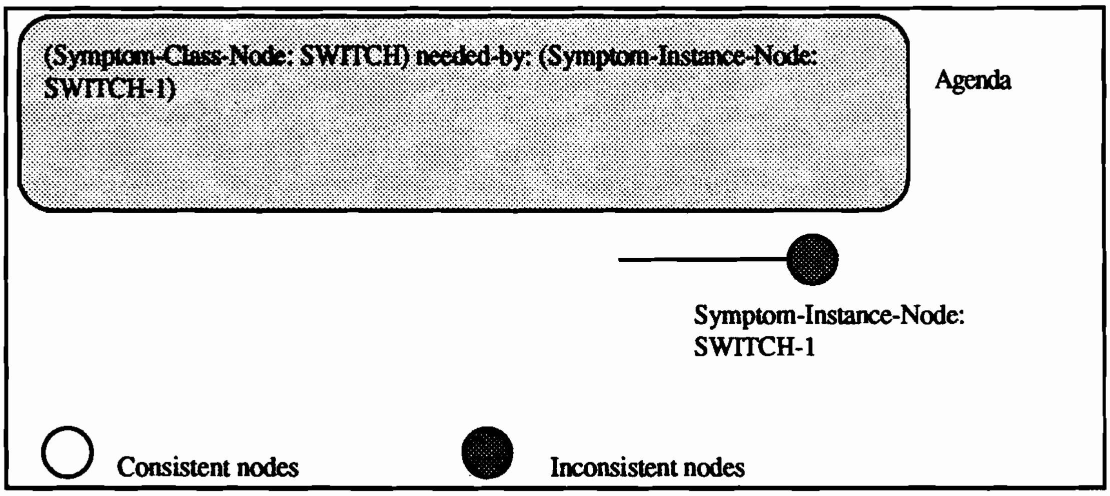

(Symptom-Class-Node: SWITCH) needed-by: (Symptom-Instance-Node: SWITCH-1)
Agenda
Symptom-Instance-Node: SWITCH-1
Consistent nodes
Inconsistent nodes

Figure A.1: Step 1 - The resulting knowledge dependency network

Step 2:	Symptom-Class	new_with_name: SWITCH with_Type: SWITCH-TYPE

What happens? A symptom class node is put in the knowledge dependency network and marked as inconsistent. An entry for the needed type is put into the agenda. Then the system finds out that the new node is already referenced in the agenda and the appropriate connections are established.

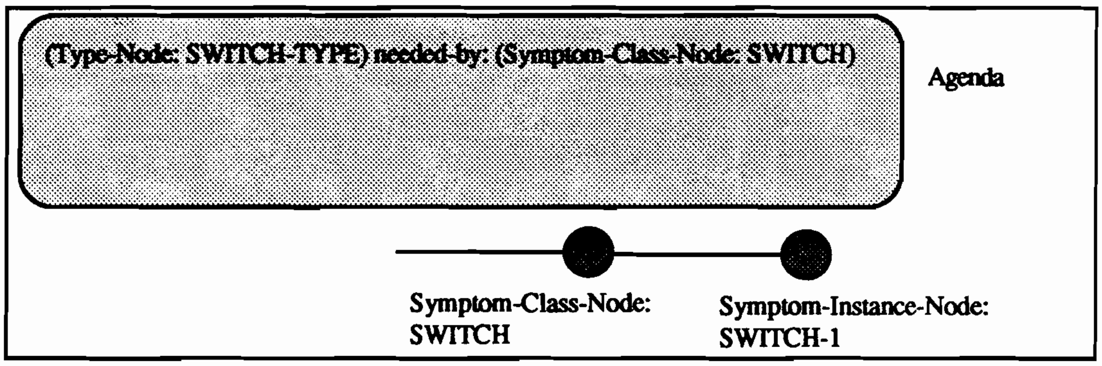

(Type-Node: SWITCH-TYPE) needed-by: (Symptom-Class-Node: SWITCH)
Agenda
Symptom-Class-Node: SWITCH
Symptom-Instance-Node: SWITCH-1

Figure A.2: Step 2 - The resulting knowledge dependency network

Step 3:	Type	new_with_name: SWITCH-TYPE values: (open closed)

What happens? A type node is put in the knowledge dependency network and marked as consistent. Then the system checks the agenda for an entry which matches the new type. This entry is removed from the agenda and the (referenced) symptom class node is marked as consistent. From this results that also the symptom instance node is consistent. All nodes enter the knowledge base.

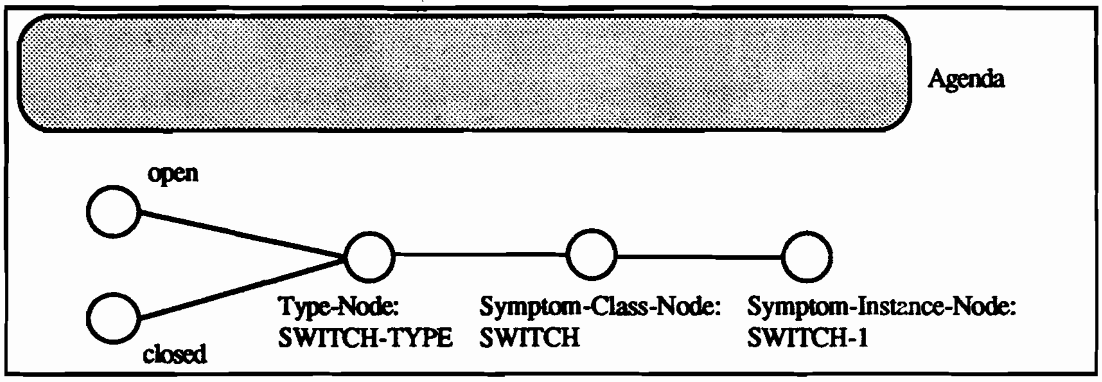


Figure A.3: Step 3 - The resulting knowledge dependency network

Step 4:	Symptom-Class	new_with_name: POWERSWITCH with_type: SWITCH-TYPE

What happens? The user tries to define a symptom class node. The system checks the inconsistency graphs and detects a contradiction to graph 2 of figure 2.6. So, the new node is rejected.

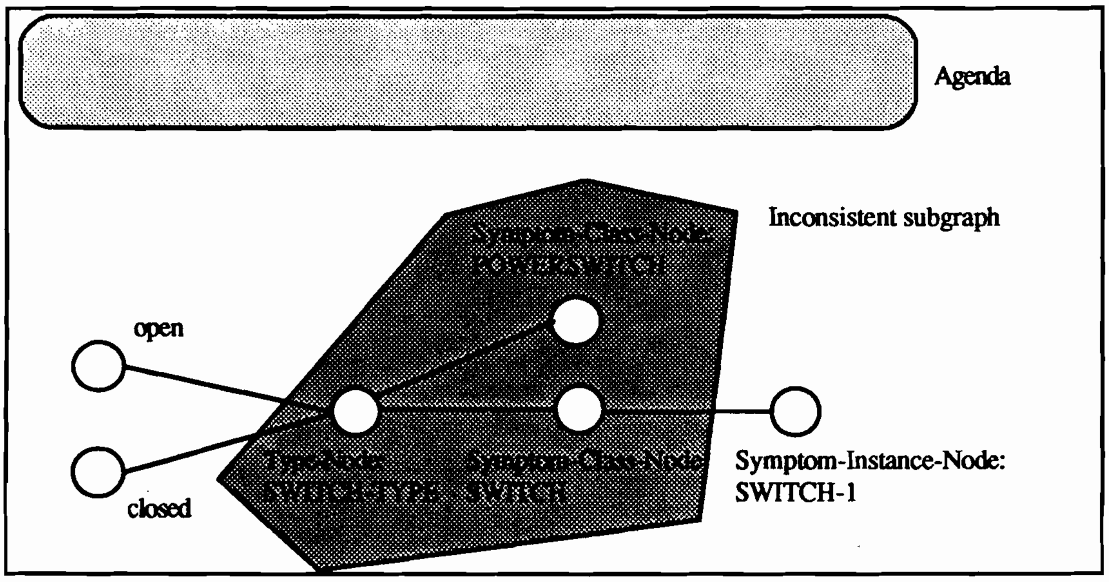

Agenda
Inconsistent subgraph
Symptom-Class-Node: POWERSWITCH
open
closed
Type-Node: SWITCH-TYPE
Symptom-Class-Node: SWITCH
Symptom-Instance-Node: SWITCH-1

Figure A.4: Step 4- The knowledge dependency network and the inconsistent subgraph

Step 5: Type new_with_name: POWERSWITCH-TYPE
values: (open closed)

What happens? The user tries to define a type node. The system checks the inconsistency graphs and detects a contradiction to graph 1 of figure 2.6. So, the new node is rejected.

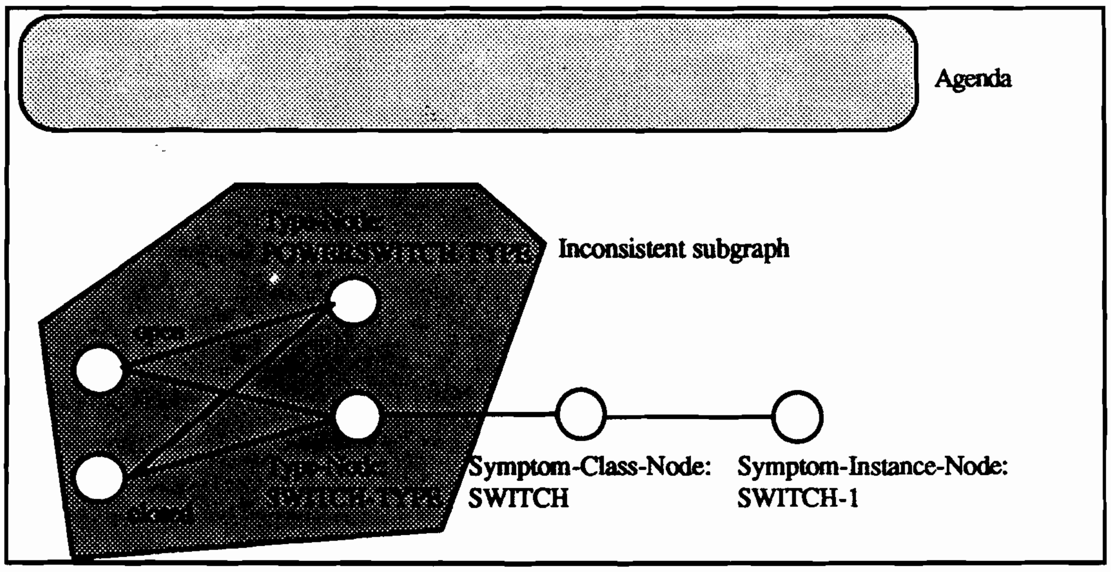

Agenda
Type-Node: POWERSWITCH-TYPE
Inconsistent subgraph
open
closed
Type-Node: SWITCH-TYPE
Symptom-Class-Node: SWITCH
Symptom-Instance-Node: SWITCH-1

Figure A.5: Step 5- The knowledge dependency network and the inconsistent subgraph

REFERENCES

Althoff, K. D. (91). A case-based learning component as an integrated part of the MOLTKE$_3$ workbench for the diagnosis of technical systems (in german: Eine fallbasierte Lernkomponente als ein integrierter Bestandteil der MOLTKE$_3$ Werkbank für die Diagnose technischer Systeme), Diss. University of Kaiserslautern, 1991 (to appear)

Althoff, K.D., Maurer, F., Rehbold, R. (90). Multiple Knowledge Acquisition Strategies in MOLTKE, in: Proc. EKAW 90

Althoff, K. D., Maurer, F., Traphöner, R., Weß, S. (90). The learning component of the MOLTKE$_3$ workbench for the diagnosis of technical systems (in german: Die Lernkomponente der MOLTKE$_3$ Werkbank für die Diagnose technischer Systeme), KI, special edition on Machine Learning, Munich: Oldenbourg Verlag, No. 1, 1991

Althoff, K. D., Maurer, F., Wess, S. (91). Case-Based Reasoning and Adaptive Learning in the MOLTKE3 Workbench for Technical Diagnosis, Technical Report University of Kaiserslautern, 1991

Althoff, K. D., Traphöner, R. (90). GenRule: Learning of Shortcut-Oriented Diagnostic Problem Solving in the MOLTKE$_3$-Workbench, Technical Report University of Kaiserslautern, 1990

Breuker, J., Wielinga, B. (87). Model-Driven Knowledge Acquisition: Interpretation Models, Memo 87, Deliverable task A1, Esprit Project 1098; 1987

Campbell, B. , Goodman, J. M. (88). HAM: A General Purpose Hypertext Abstract Machine, Communications of the ACM, July 1988, Vol. 31, No. 7

de Kleer, J. (86). An assumption-based TMS, Artificial intelligence, Vol. 28, P. 163-196, 1986

Doyle, J. (79). A truth maintenance system, Artificial intelligence, Vol. 12, P. 231-272, 1979

Jansen, B., Compton, P. (89). The Knowledge Dictionary: Storing Different Knowledge Representations, in; Proc EKAW 89

Maurer, F (91). CAKE: Computer-aided Knowledge Engineering, Proc. of the IJCAI-91 Workshop on "Software Engineering for Knowledge Base Systems"

Maurer, F. , Ruppel, A. (90). Learning of diagnostic strategies with neural networks in the MOLTKE 3.0 expert system toolbox (in german: Lernen von Diagnosestrategien mit neuronalen Netzen in der MOLTKE 3.0 Expertensystem-toolbox), Technical Report University of Kaiserslautern, 1990

Morik, K. (86). Acquiring domain models, in: Knowledge Acquisition Tools for Expert Systems, Academic Press, 1988

Musen, M. A., Fagan, L. M., Combs, D. M., Shortcliff, E. H. (86). Use of a domain model to drive an interactive knowledge-editing tool, in: Knowledge Acquisition Tools for Expert Systems, Academic Press, 1988

Nökel, K. (89). Temporal Matching: Recognizing Dynamic Situations from Discrete Measurements, in: Proc. IJCAI 1989

Nökel, K. (91). Temporallly Distributed Symptoms in Technical Diagnosis, Springer Verlag, Heidelberg/Berlin/New York, 1991

Rehbold, R. (89). Model-Based Knowledge Acquisition from Structure Descriptions in a Technical Diagnosis Domain, Proc. Avignon 1989

Rehbold, R. (91). Integration of model-based knowledge into technical diagnostic expert systems (in german: Integration von modellbasiertem Wissen in technische Diagnostik-Expertensystem), Diss. University of Kaiserslautern 1991

Richter, M.M. (89). Principles of artificial intelligence (in german: Prinzipien der künstlichen Intelligenz), Teubner Verlag, 1989

Richter, M. M. (92): Das MOLTKE-Buch (in German), (to appear)

van Someren, M. W., Zheng, L. L., Post, W. (91). Cases, Models or Compiled Knowledge; a Comparative Analysis and Proposed Integration, in: Proc. EKAW 1990