Multiple Knowledge Acquisition Strategies in MOLTKE [1]

Klaus-Dieter Althoff, Frank Maurer, Robert Rehbold
University of Kaiserslautern
Dept. of Computer Science
P.O. Box 3049, D-6750 Kaiserslautern
West Germany

Abstract

In this paper we will present a design model (in the sense of KADS) for the domain of technical diagnosis. Based on this we will describe the fully implemented expert system shell MOLTKE 3.0, which integrates common knowledge acquisition methods with techniques developed in the fields of *Model-Based Diagnosis* and *Machine Learning*, especially *Case-Based Reasoning*.

1. Introduction

When starting a real world expert system project knowledge acquisition is the bottleneck. A main reason for this is the gap between the languages of the domain expert and that used for the implementation by the knowledge engineer. This gap reflects the difference between the cognitive models of the application area the two involved persons have. According to the KADS-group to overcome these shortcomings a conceptual model of the domain has to be developed [1]. Then this conceptual model has to be transfered into a design model.

We think that developing a conceptual model has a few drawbacks:

- The knowledge of an expert includes static and dynamic parts. Static knowledge can be represented in semantic networks whereas it is not easy to express the dynamic parts which describe how a problem is solved. The interpretation model often lacks a clear (i.e. formal) semantics.
- If people express knowledge without having an inference engine in mind the transfer of the developed representation into a running expert system may be hard. It becomes harder the bigger the gap between the conceptual model and the language of the used shell is.

---

1 The work presented herein was partially supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314: "Artificial Intelligence- Knowledge-Based Systems", projects X6 and X9.

- Primitives of particular technical application domains are easily found, e.g. in diagnosis one always talks about symptoms, faults and tests.

Our approach to reduce the knowledge acquisition bottleneck is to describe a design model which reflects the primitives of the application domain. It has to include an inference engine for the knowledge. The task of knowledge acquisition is to fill the templates of the shell with the expert´s knowledge. If the primitives really reflect the concepts of the application area the communication between the knowledge engineer and the expert is improved because they speak the same language.

Starting from a real world application we developed a design model for technical diagnosis, implemented its basic vocabulary and an interpreter. The MOLTKE 3.0 Shell, which is described in the next section, reflects the application domain concepts like symptoms, tests and failures. Building up on this kernel more sophisticated tools were developed:

- A compiler which generates the core of an expert system for diagnosis out of a deep model of the technical device.
- Case-based reasoning techniques to enlarge the knowledge acquisition process.

As techniques of three general approaches (deep modelling, machine learning and manual knowledge acquisition) are fully integrated, the MOLTKE system is able to use multiple sources of knowledge.

2. A Design Model for Technical Diagnosis

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

Diagnosis = Classification + Test Selection.

The state of a technical device is described by a list of *symptoms* (where symptom means something that is measurable, e.g. voltage). A symptom can be low level, e.g. the real voltage at a special measuring point, or more abstract, e.g. "voltage is too high". Therefore a shell for diagnosis has to include mechanisms for data abstraction. Furthermore there is a need for expressing time-dependent symptoms, so called temporally distributed symptoms (TDS) (the integration of TDS in MOLTKE 3.0 is described in [2]).

At any point in time a symptom can have only one *value*. A list of actual symptom values is called a *situation*. In a situation a symptom may also be unknown. A failure in a technical device can be expressed in terms of particular values. Therefore, a *fault hypothesis* is correlated with a set of symptom values. If the actual situation matches these, the failure is established.

Often some symptom values determine others, e.g. if the light in a room is on one can infer that the wires are working. For that reason an expert system must handle relations between symptom values. Moreover this reduces the search space because the determined symptoms need not to be asked.

The search space for diagnosis is described by a graph where nodes are situations and arcs between situations are labelled with a test. A *test* determines the value of symptoms (by asking the user or by getting sensor information). Tests are executed if according to the actual situation no failure can be proved. In technical diagnosis the processing of symptom values must be both demand-driven (the system decides what to measure next) and data-driven (asynchronous incoming data is processed immediately).

Diagnosis is a process: a situation where no failure is proved must be transformed by executing tests into one where the description of a fault matches the actual symptom values. A *strategy* describes the order of testing and should reflect the experience and heuristics of diagnostic experts as e.g. service technicians.

Modularisation always is a means of software engineering which improves the maintainability of a program. Expert systems shall use the advantages of modularisation. Therefore, knowledge about failures and about strategies shall be separated for reasons of a horizontal modularisation. Vertically the expert system is split in a hierarchy of rough, intermediate, and final diagnoses. The modularisation helps to integrate the knowledge of multiple experts.

3. The MOLTKE Project - An Overview

MOLTKE means MOdels, Learning and Temporal Knowledge in Expert systems for technical diagnosis. Within the MOLTKE project (see e.g. [3,4]) we developed an expert system toolbox - the MOLTKE system - and several applications. Figure 1 gives an overview of the MOLTKE system. The shaded part in the center is the kernel, the MOLTKE 3.0 shell. The model component on the left hand side and the case-based reasoning part on the right hand side are described later in this paper.

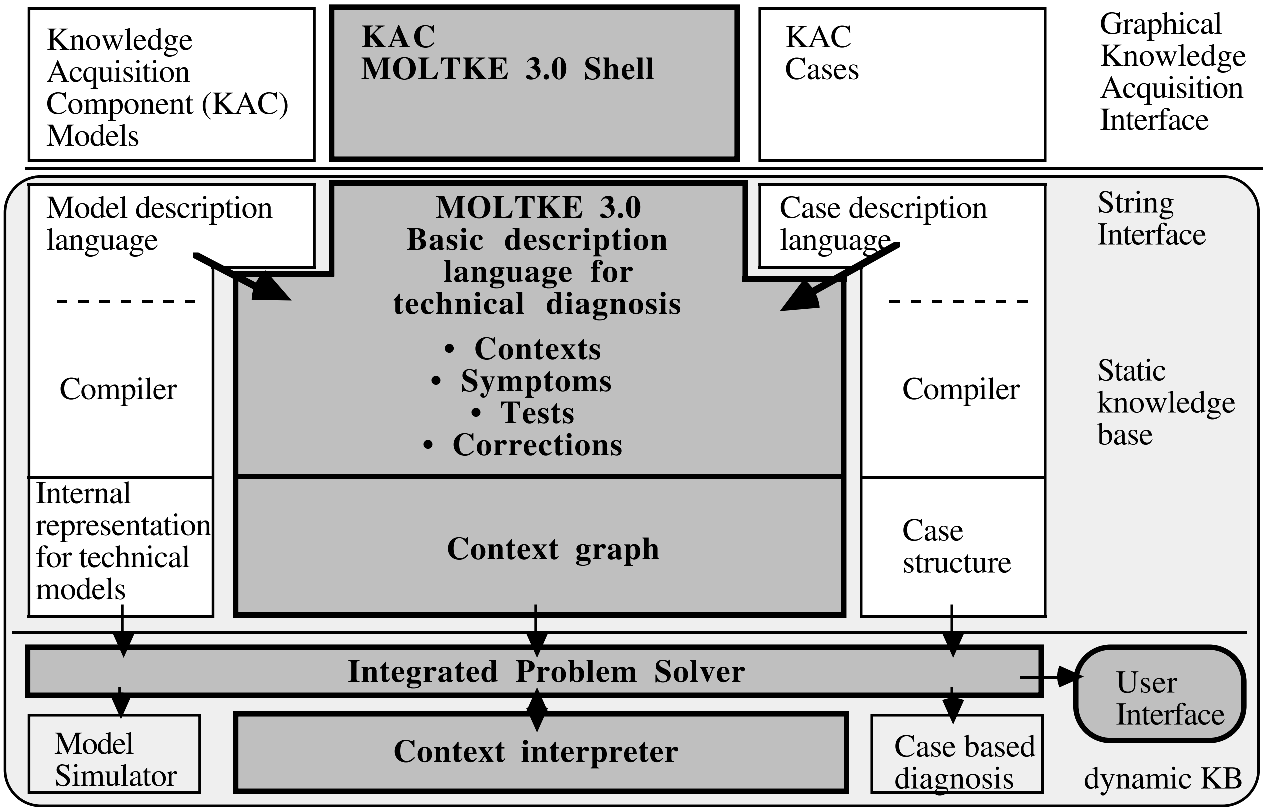

Figure 1: The MOLTKE System

The MOLTKE system is a modularized toolbox for the development of expert systems in the domain of technical diagnosis. It integrates common knowledge acquisition techniques with deep modelling and machine learning. For the interpretation of a knowledge base several interpreters were implemented which allow to integrate different sources of knowledge (see chapter 8).

4. The MOLTKE 3.0 Shell

4.1. The Static Knowledge Base

The static knowledge base represents the knowledge about the technical system. Here we describe its basic vocabulary:

A *symptom class* relates a name to a list of possible values (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 possible value list in the corresponding class.

The set of all symptom instances is called a *situation*. In the context of predicate calculus the actual situation is the base for the interpretation of a *language of formulas.* It stores the bindings of the variables[1]. We use a three-valued logic with *true, false* and *unknown*. For reasons of an efficient evaluation all formulas are stored in network similar to the Rete-Network of OPS5.

A *test* ascertains the value of one or more symptom instances[2]. Every test has an associated precondition (e.g. for specifying that a test can only be executed if the technician has a particular level of skill we would use a precondition like (= skill level high)). A test is only executed if the corresponding precondition is evaluated to true. 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 OUT30 test)).

To express relations between symptom values *shortcut rules* are used (e.g. (if (= lamp room on) then wires := working)). To express uncertainty every rule may have a list of associated qualitative certainty factors called determination factors[3]. Uncertain shortcut rules are called *partial* in opposite to the *total* ones which are certain with respect to the underlying deep model. These rules are called shortcuts because they abbreviate the diagnostic process, i.e. symptom values which are determined by shortcut rules need not to be asked anymore.

Contexts are one of the means for modularisation in MOLTKE 3.0. 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. Any context contains a set of ordering rules which locally prescribe the strategy of testing. Additionally, a context includes a set of shortcut rules. Every context contains a slot for a *context interpreter* which determines the next symptom to be tested. The default context interpreter uses the ordering rules for test selection[4].

A correction describes what has to be done when a special fault occurs. Up to now a description how to repair the device is given to the user. For monitoring tasks this may easily be extended to manipulating control units.

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*.).

---

1 Every symptom instance is a variable in the calculus.

2 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.

3 The processing of these factors is described in section 7.5.

4 We developed neural network associators and model-based components which can replace the default context interpreter.

4.2. The Interpreter

The processing of a knowledge base is done by a global interpreter which is easy to adapt and to maintain because it is organized in small modules. The default interpreter realizes an establish-and-refine-strategy (see figure 2).

```
Algorithm of the default interpreter:

actualContext := Root of context graph;
while actualContext not a final diagnosis do
        process all shortcut rules with a true precondition;
        rule := select-an-ordering-rule(actualContext, actualSituation);
        value := test-the-selected-symptom(rule);
        propagate the symptom value through a rete-network;
        if the actualContext has refinements with a true precondition
                then actualContext := this refinement;
        if all refinements of the actual context are false
                then backtracking;
```

Figure 2: The algorithm of the global interpreter

The diagnostic process walks 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 is reached the system stops.

The system is able to reset a symptom value to unknown and to reject all inferences by shortcut rules based on its old value. This facility is used when the user or the system retracts uncertain symptom values.

Sensor data are propagated through the formula network when they arrive[1]. Every time the system asks for a special symptom value the user is able to enter the value of another symptom instance. According to the actual state the system determines the actual context.

5. Knowledge Acquisition with the MOLTKE Kernel

The MOLTKE 3.0 kernel includes a comfortable knowledge base browser based on the facilities of the Smalltalk-80 system. The browser combines a menu-oriented interface with different editors for symptoms, contexts, tests, etc. Editing an object means filling a template. In a graphic window the context graph can be edited. The editors include syntax- and type-checkers. If other objects are referenced[2] their availability in the knowledge base is tested.

While editing a knowledge base the system is always able to run. If in a context no ordering rule can fire, the system asks for a symptom value which is used in the preconditions of its refinements. We will extend this facility by asking the user what symptom shall be tested and then generate an appropriate ordering rule for the context.

For testing a knowledge base trace- and debug-facilities were implemented. The user is able to set or reset arbitrary symptom values and inspect the state of all formulas and the formula network according to the actual situation. At the moment we are developing a consistency checker for the knowledge base.

The runtime environment is menu-oriented. Additionally the user is able to enter symptom values by mouse-clicking in some graphics of the technical device. The kernel includes an editor for these graphics.

---

1 So a data-driven updating of the actual situation is possible.

2 For example symptom instances referenced in formulas.

6. Knowledge Acquisition Using Deep Models

Complex technical devices often have a common property: Due to task customization and rapid further developments only few fully identical instances of a given type are built. This makes it difficult to find somebody who is really an expert for the maintenance of this special machine, especially if the type is relatively new. Service technicians troubleshooting such a new machine cannot rely on machine-specific heuristics, but have to use general technical knowledge, a detailed description of structure and behavior of the machine and a general understanding of the functionality of the device to find any faults.

Since diagnostic expert systems are expected to be able to assist in troubleshooting right after the release of a new type of machine they have to use structural, behavioral and subpart information first, too. As time goes by technicians learn more about a special machine type through the faults they are confronted with during their work. This learning from cases has to be embodied in a sophisticated expert system too, giving the need for a knowledge representation that equally suits the causal (model-based) as well as the heuristic (case-based) knowledge. Furthermore an existing expert system should be easily adaptable to a slightly changed machine, e.g. from a new series or with different optional subdevices (cf. [5]).

6.1. Causal Knowledge and Deep Models

We will use the word "causal" in this paper to distinguish things that can be traced back directly and with absolute certainty to structure, behavior and function of a single concrete physical device from those that do not, either since they lack a clear correlation to the physical facts (these we will call "heuristic") or since they depend on many instances of a device and are not certain (called "statistic"). Thus causal knowledge in a technical domain consists of all the facts on structure, behavior and function of the device together with information on how connected parts interact and can be considered synonymous to the term "deep knowledge". From the causal knowledge of a device a representation can be build which in this chapter we will call a deep *model*, or just a model.

6.2. The Task: Getting the Causal Knowledge into the System

A possible approach to get some of the causal knowledge about a device into the expert system could be to have an engineer (i.e. someone with sufficient "technical common sense", not necessarily an expert) look at the design plans and diagrams of a concrete machine and write down the information in form of failure detection flowcharts which are then translated into rules of different types. The order in the flowchart would be based on a priori failure probabilities and directly influence the ordering and priorities of the rules. When we tried this way of knowledge acquisition for the first application in project MOLTKE, which was an expert system for the diagnosis of CNC[1] machining centers, it turned out that it was quite time consuming, since the process required several iterations and the final flowcharts were by no means complete (with respect to the underlying descriptions). Worse, new machine series or, even more, types required much of that work again, since most diagrams had to be revisited to check for changes (e.g. new components with new failure probabilities or even new failures).

This caused us to build a tool (MAKE = Model-based Automatic Knowledge Extractor) [6] to do the work of that engineer. MAKE gets diagrams (i.e. the structure) of the machine and uses its built-in knowledge of electrical and mechanical engineering (i.e. its "technical common sense") - e.g. the behavior of typical components - to form a (deep) model of the machine and to derive rules for test suggestion and determination of yet unmeasured symptoms (shortcut rules) from this model.

---

1 Computerized Numerical Control

Furthermore MAKE is able to organize the rules into contexts and to create corresponding preconditions for them. We call such an expert system without any machine specific heuristic knowledge *basic expert system.*

An important point is that the diagrams can be entered by anyone; there is no need for the special knowledge of an engineer to do that. An engineer (the expert) is needed only to provide information about the typical atomic parts in the descriptions, e.g. valves, switches, contactors, hydraulic pistons. Some statistic knowledge about a priori failure probabilities and additional information on measurement costs have been integrated into the model to allow a more sophisticated test selection; thus the model is no longer purely causal, but contains other knowledge, too[1].

Note that the application domain of the whole MOLTKE system (and therefore of MAKE) is not as simple as digital circuits, the usual example domain for model-based diagnosis.

6.3. Available Knowledge when the First Machine is Built

As heuristic knowledge of a new machine will not be available when the basic expert system is to be built, one has to restrain oneself to the information available at that point of time:

- knowledge about the primitive (and maybe complex[2]) parts (components) used
  * possible connections (ports), i.e. how the component may be connected to others
  * behavior, i.e. how the component is expected to react to changes in connected components
  * misbehavior(s), i.e. what typical faults may occur, how they change the expected behavior and how common they are[3]
- knowledge about the connectivity of the components, i.e. a structure description
- knowledge about measuring costs at certain points[4]
- knowledge about the intended functionality of the machine as a whole, i.e. which set of inputs should cause which set of outputs.

The "birth" of a basic experts system then consists of four major steps:

1) An expert builds up a library of the primitive (and maybe common complex) components that are used. This task needs an expert since it should be done as precisely and completely as possible for the sake of the resulting expert system. Of course the library can be used for many different machines thus an expert is not needed for every single machine type or series.

2) A person with technical skill - at least he/she shall be able to read, understand and reproduce technical structure descriptions - is needed to enter the connectivity of a given machine, i.e. to build up the structural model. He/she will select parts from the library and build them together to more complex ones finally describing the whole machine. A graphical editor that allows to mimic the technical diagrams helps to accomplish this step. Note that this task is independent from the availability of a diagnosis expert if the library (cf. step 1) completely covers all components used.

---

1 However, all the knowledge (information) used by MAKE is available when the first machine of a type is delivered to the customer, i.e. no heuristic information on the special machine type is used.

2 Often information will be available on more complex subdevices if they have already been used before in other machine types. However, the relevance of this information is not always clear (and is therefore subject to be changed by a mechanism to adapt it from e.g. cases).

3 This information usually comes from the manufacturer of the primitive component. However, the a priori failure rate can vary dramatically with the position of the part in the machine. Learning from cases can adjust either the heuristic information embedded in the model or the diagnostic system itself to such special situations.

4 All complex machines have sensors etc. attached to them and provide designated measuring points where data can be gathered inexpensively.

With a fully computerized design, developement and production system (CIM[1]) it should be possible to get the connectivity information immediately from the design data.

3) The designer of the machine can specify the desired functionality in form of input/output behavior. While this information can be left out (MAKE could simulate all input/output behavior from the given knowledge itself) it is helpful to cut down all possible (and maybe unintended or unused) behaviors to the intended and used ones. When dealing with machines with a control component, e.g. a CNC, the CNC programs give information on what functions the machine is expected to perform.

4) Finally, the MAKE system transforms the model of the machine (put together from the library and the connectivity) with help of the desired input/output behavior into a knowledge base for the expert system shell MOLTKE.

6.4. The Model

We use a component oriented, hierarchical qualitative model of the machine. What is considered an atomic component is determined by the granularity of the used diagrams and the exchangeability of the parts (e.g. since relays are completely replaced if found faulty, there is no need to model the parts of a relay). Information on typical *primitive* (atomic) components like valves, switches etc. is stored in a library. Using these parts, new assembly groups (called *complex* components) can be built and so on; this way a hierarchical model is built up where the top component represents the entire machine. The general description of a component is found in the component class, its instances represent the concrete components. Connections can be internally modeled as components, but need not to be entered explicitly, since they can be added automatically (e.g. if the user connects two hydraulic ports the system automatically assumes a pipe between them).

A component class stores the following knowledge:

- name of the component type
- ports to other components (optionally with test costs)
- possible internal states (optionally with test costs)
  Internal states can be made available to other components using them as ports.
- behavior of the component
  The behavior is given either in form of tables or by rules: the if-part contains predicates on port values and states that allow to conclude the value of some other port or state in the then-part. These tables/rules represent the constraints the component sets up between its ports and states. Note that components do know about their function, as the "no-function-in-structure" principle[2] is deliberately abandoned to enable the generation of better causal rules from the model.
- subparts and their interconnections (only if the component is non-atomic)
- typical malfunctions with name and effects (if available)
  These typical malfunctions model the behavior of the component when a common failure occurred and enable the system to reason faster. There is always the possibility to fall back to the total suspension of the constraints of the component ("unknown failure"), which is the usual approach of model-based troubleshooting[3] [7].

---

1 Computer Integrated Manufacturing

2 This principle is often demanded in the literature on model-based reasoning and qualitative simulation. Its generality, however, while being useful in the prediction of unknown behavior, disallows us some interesting conclusions that we can draw since we already know the function of the part.

3 Note, however, that shortcut rules will only be 100% correct if the malfunctions listed here are the only ones possible. If "unknown failure" is among the possible misbehaviors of a component class, no certain shortcut rules

- a priori probability of failure (if available)

An actual component is an instance of its class which additionally knows its name, location, neighbors (i.e. which of its ports is connected to which port of which (possibly) other instance), names of its subcomponents (if any) and position in the diagram. Note that each port of each component instance later becomes a symptom in the basic expert system generated by MAKE.

Component classes are immediately cross-checked: topology, states and behavior must be fully specified, all ports of subparts must be connected by connections of the correct type[1] and states must appear in the behavior. When actual parts (instances) are entered, they are cross-checked too to ensure that every subpart class exists and is of proper type and that the connections to the neighbors are ok. Currently a special editor to support the creation of component classes and instances is under development together with a graphic tool to enter the structure of the machine.

6.5. Functionality Description

The expected functionality of a device has to be entered as relations between its input and ouput ports. When dealing with computerized numerically controlled (CNC) machines (e.g. MOLTKE's first application: CNC machining centers) even more than these relations can be taken from the CNC programs. These programs do not only provide us with input/output information but usually also contain data on failures the CNC is able to detect. When the CNC detects such a failure it stops the machine and displays a failure number that provides us with first information about where the malfunction happened.

These failure messages are issued if some feedback signal to a started machine order misses or is wrong (e.g. a limit switch did not operate within a given time)[2]. This information allows to focus the attention to a certain part of the machine and is therefore a valuable starting point for the diagnosis. Each failure is associated with a certain machine subcycle that performs a special operation starting with some control system output signals and ending with some feedback input signals. It is possible to conclude from the model which valves, switches etc. are related to these IO signals and are therefore under suspect.

When no CNC is used MAKE has to simulate the device itself to be able to detect deviations from the expected input/output relations and to cross-check the correctness of the functionality specified by the designer. In these cases important inputs and outputs have to be measured before the focus of attention can be moved to a certain part of the machine.

If no functionality at all is specified MAKE must generate all possible input combinations and simulate them through the model to get the respective expected output values. The input/output relations thus generated can than be used as if entered by the designer.

6.5.1. Building Contexts from the Model

The best way to choose contexts is to ask the experts; but since we want to build the basic causal expert system without their help, we have to build the contexts ourselves. Naturally, automatically generated contexts may be not as good as those from the experts, but our experience with the manually chosen contexts showed some typical patterns that will most often fit. Each context (intermediate diagnosis) leads to several more special diagnoses that are its subcontexts. Leaf

can be generated from the model since it is impossible to make any conclusions about the behavior of the component. Of course, a case based reasoning mechanism still can derive uncertain shortcuts in such a situation.

1 Ports have types and directions to ensure proper modeling.

2 This shows an additional advantage of diagnosis of a CNC-controlled machine: The CNC observes the inputs and outputs of the machine automatically, so we already know these values when the diagnosis starts.

contexts correspond to final diagnoses. An example for a part of a context heterarchy is given in figure 3.

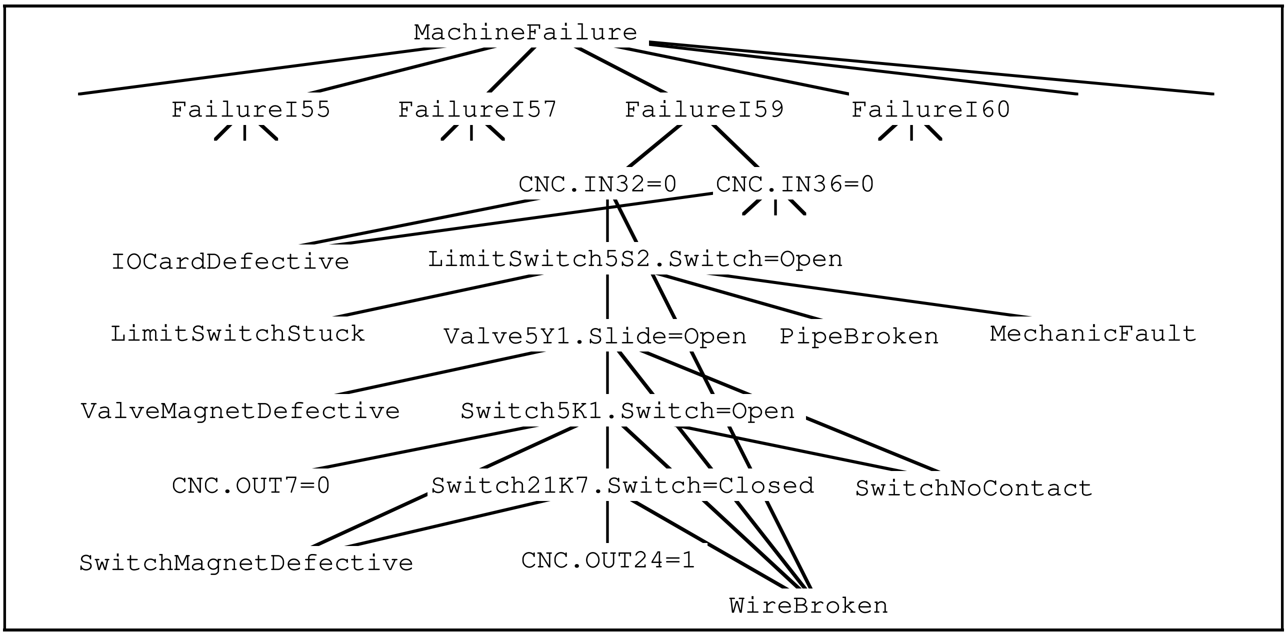

Figure 3: Example of a context heterarchy

The most abstract context and root of the context heterarchy is a context called `MachineFailure`. Its first level of subcontexts is induced by the possiblities of how the input/ouput relations given in the functionality description can go wrong. When a CNC with a integrated diagnostic system is present, this is the level of control system failure messages, i.e. each failure number has its own context (e.g. `FailureI59` in fig. 3). Each of these top level failure contexts now gets subcontexts of its own for each (control) system input that could have been responsible for that failure (e.g. `IN32=0`). Since the same fault could lead to different top level failures, depending on the machine cycle it occurred in (i.e. the inputs), the contexts are organized in a heterarchy instead of a hierarchy to avoid multiple instances of the same context.

MAKE builds a context class C for each possible faulty value V on outport O of component class Co (primitive and complex). The name of C is automatically generated as "Co.O=V" (e.g. `ValveSelection.Switch=Open`). It may seem as if this method creates a very large amount of contexts, but since contexts are only generated for those outports and values that actually force faulty output values in the whole machine, not all combinations will be created. The number of contexts generated thus depends not only on the number of different components and the number of their outports and value ranges, but especially on the constraints in the expected input/output relations given in the functionality description.

There are three basic strategies how to decide from the model what subcontexts a given context gets; each of them constructs a precondition for them that is both necessary and sufficient to establish the diagnosis the context represents:

1) *hierarchical strategy*

This strategy works strictly top-down according to the hierarchical model of the machine, i.e. it first makes sure that the fault occurred inside a component Co (by testing that Co's inputs are as expected while at least one of its outputs is not, e.g. outport O has the unexpected value V) and then switches to the context C for Co. The precondition for entering C thus becomes "all inputs of Co have expected values and output O has unexpected value V". All contexts of the form "$SCo_i.O_{i,j}=V_{i,j}$" become subcontexts of C, if SCo_i is a subcomponent of Co and value $V_{i,j}$ at SCo_i's outport $O_{i,j}$ can cause value V at Co's outport O.

2) *sequential strategy*

Unlike the hierarchical strategy the sequential one does not use the hierarchical structure of the model. Instead, it traces the simulated behavior from the outputs of the machine back to its inputs on the lowest component level thereby necessarily detecting the malfunctioning primitive component. The precondition of a context for the sequential strategy therefore consists only of the fact "Co's outport O has the unexpected value V". All contexts of the form "$Co_i.O_{i,j}=V_{i,j}$" become subcontexts of C, either if Co_i is a subcomponent of Co, Co_i's outport $O_{i,j}$ is directly connected to Co's outport O and $V_{i,j}$ = V, or if Co_i's outport $O_{i,j}$ is directly connected to one of Co's inports and a value of $V_{i,j}$ at that inport can cause value V at Co's outport O.

3) *mixed strategy*

This strategy is a mixture of the hierarchical and the sequential ones. It performs backward tracing like the sequential one on a given level, but moves down in the hierarchy only if it can make sure the failure lies within the subcomponent. As we found out this is the strategy the typical service technician seems to follow: He/she traces the faulty output backwards on the top level until he/she comes to a component with correct inputs and faulty output. He/she then moves one level down in the hierarchy and again performs backtracing on that level. The precondition of a context in the mixed strategy therefore is the same as in the hierarchical case, while the subcontexts are selected like in the sequential case.

6.5.2. Construction of Shortcut Rules

A shortcut rule tries to establish facts about yet unmeasured port values from already acquired data. Deriving them from a model requires knowledge about the way connected components interact. Shortcut rules derived from the model are total (certain) if all possible failures of components are modeled. Since these rules cannot assume any component to work correctly, too many different possible failures make it impossible to derive any shortcut rule at all. If an unmodeled fault occurs the shortcut rules may conclude wrong port values, but this kind of failure can be found by direct use of the model, e.g. by constraint suspension [7] or similar techniques which, however, will be quite costly. Alternatively, this is also a situation where case-based reasoning can be used.

We have different contexts for the same component with state, one for each possible output value, but the shortcut rules are the same for all of them, since they are only based on the model of the working device. We derive the shortcut rules for a component as follows:

i) Each behavior rule is copied as a shortcut rule *iff* there is no modeled failure that produces an unexpected (different) output[1] on the same inputs.

ii) The inversion[2] of each behavior rule is copied as a shortcut rule *iff* there is no modeled failure that produces the same output on different inputs.

Inside the context, the shortcut rules of each component are collected together with some rules that state the equality of connected ports.

<u>EXAMPLE:</u>

Given `(currentIn = X) -> (currentOut = X)` as the behavior rule for `Wire` and `(Broken (currentOut = 0))` as the failure of `Wire`, we can conclude the following:

---

1 Outputs here are ports or states that appear on the right side of a behavior rule; inputs appear on the left side.

2 By inversion we mean the rule that is created by swapping the precondition and the conclusion of the original rule.

Since for all $X \neq 0$ `Broken` produces a different (unexpected) output (i.e. 0) on input X than the behavior rule, i) can only be used for X=0. Thus i) leads to the following shortcut rule:

```
(currentIn = 0) -> (currentOut = 0)
```

`Broken` produces output 0 for every input, not just on 0. Thus, according to ii), we must exclude X=0 from the inversion of the behavior rule and can conclude the shortcut rule

```
(currentOut = X) (X ≠ 0) -> (currentIn = X)
```

In the last two sections the question may have arisen why we first model the component with its correct and faulty behavior and then mechanically derive contexts with preconditions and shortcut rules out of it instead of writing them down immediately. Beneath the fact that the representation of the connectivity would be more difficult and that consistency checks would have to be reduced the chosen representation is much clearer, easier to produce and can be used for other tasks too (e.g. construction of ordering knowledge - see next section).

6.5.3. Construction of Ordering Knowledge

Our augmentation of the model with failure probabilities and measurement complexity allows us to produce a list of applicable tests ordered according to some chooseable criteria for any situation. To evaluate all possible testing points inside the component a context belongs to (i.e. all the ports of the component and its subcomponents) we build up a dependency graph containing the expected values of these testing points as well as the already measured data[1]. If measured and expected value of some port (node) inside the graph agree, all testing points downward from there can be discarded as long as there is no other connection between them and the root of the graph. If expected and measured value differ this point (node) becomes the new root of the graph (thus cutting all testing points above it) since we found a port nearer to the machine inputs where things went wrong[2]. As can be concluded from this, internal nodes of an existing graph only contain yet unmeasured ports (testing points).

How do we select the next measurement from a given graph? First we examine the information gain we would get if we knew the value of an internal node. This information gain is measured by looking at the resulting graph after each possible outcome of the measurement. The resulting graphs are weighed with the a priori failure probability of the components they include and the resulting value is set off against the difficulty to actually measure the value. How exactly the weights are is determined by the chosen strategy, e.g. whether lots of easy measurements are preferrable to only few but difficult ones. An optimal measuring point would be easy to measure and would split the graph into two equally weighted halfs.

7. Knowledge Acquisition Using Empirical Cases

7.1. Motivation and Overview

As the task of an expert system which builds up on the MOLTKE toolbox is to simulate the diagnostic problem solving behavior of the respective expert, it is a necessity to represent at least the dynamical parts of the expert´s problem solving behavior which have a great influence on his/her problem solving performance. Especially his/her learning behavior - with respect to the diagnostic task, of course - is of great importance here. The pragmatic solution within the MOLTKE project is,

---

1 Root of this graph is the outport O of the component that showed the wrong value.

2 If we would allow multiple faults, we could not cut the graph this way.

as a first approximation, to distinguish between two basic learning strategies of the expert. The motivation for this is a real world application´s point of view. Therefore it is sufficient for our purposes here. The expert´s basic strategies are:

- *Learning from examples*

  Examples are a kind of problem solving traces which form the expert´s problem solving experience. The expert learns while memorizing such examples and using them as a positive or negative feedback for enhancing his problem solving knowledge.

- *Analogy*

  The expert uses analogies between the given actual problem and known examples to guide and focus his problem solving process. During this process the expert learns new and/or better examples.

Both abilities (in regard of the given task) are acquired during the expert´s normal work. The goal within the MOLTKE project is to utilize both techniques for solving complex problems. The system therefore has a component that memorizes the "learned" examples[1] (which we call cases) and applies analogical reasoning techniques for solving problems based on these concrete cases. The underlying basic hypotheses of our approach are that learning by memory adaptation and analogical problem solving are fundamental techniques which experts (and, as we believe, human beings in general) apply during problem solving. Thus, using them as the system´s central mechanisms maximizes its transparency with respect to the expert´s learning behavior, increases the user´s acceptance concerning the system and decisively improves the knowledge acquisition support.

In this connection the MOLTKE project concentrates on two points of emphasis. The first one is the direct interpretation of the learned cases which allows the application of analogical problem solving as the system´s central mechanism. For further details see [8,9]. Secondly, we stress the usage of an adaptation and extension of the determination-based analogical reasoning approach of [10]. Determinations[2] are similar to MOLTKE´s shortcut rules in the sense that the latter are compilations of analogical inferences which are justified by determinations. Thus, from a simplifying point of view a shortcut rule (if *situation1* then *symptom2* := *value*) can be interpreted in such a manner that situation1 *determines* symptom2. The latter approach enables analogical inferences which are justified with certain restrictions, as the main problem is the acquisition of such determinations/ shortcut rules. Our answer is the automatic generation of total shortcuts from the deep model of the underlying technical system (by the MAKE system, see chapter 6) and of partial shortcuts from the empirical case base using explanation-based learning (by the GenRule[3] system).

The next section gives an overview of the representation(s) of empirical knowledge in MOLTKE and its (respective) principal organization. The generation of partial shortcut rules based on an algorithm presented in [11] is discussed in 7.3. Then our approach of extracting ordering information from the empirical cases is introduced. We complete this chapter with a description of the kernel system´s capabilities for the handling of partial shortcut rules.

7.2. Representation of Empirical Knowledge

In MOLTKE empirical knowledge is represented by a collection of examples which we call cases. The respective knowledge sources could be documentation material like service reports, test bench protocols, and test cases or the expert´s remembered experiences. Cases are not only well suited for the representation of such knowledge, but also for the modelling of vague information. In case of the application on CNC machining centers this affects relations of machine faults to symptoms

---

1 Learning by memory adaptation, i.e. storing and updating of individual experiences and statistical information.

2 For a definition of determinations within MOLTKE see [11].

3 <u>Gen</u>erator of empirical MOLTKE <u>Rule</u>s

which depend on smelling, hearing, seeing, feeling, or the geometry of workpieces. From an abstract point of view typical cases can be described as follows:

diagnostic case	**strategy case**
symptom1 = 1	symptom1 = 1
symptom2 = 0	symptom2 = 0
symptom3 = 2	symptom3 = 2
symptom4 = 3	symptom4 = 3
-----------------	-----------------
diagnosis1	symptom5

The part above the line is the description of an empirical situation[1] in the form of a list of symptom values. The part below the line is the solution of the case, i.e. the (empirical) result of the described situation. We distinguish between diagnostic and strategy cases. For the first type, the result of the situation has been to make a diagnosis, for the second, to ascertain a symptom. The diagnostic cases are used for support in the classification task, the strategy cases for the selection of tests. As the symptom values within the description of the case situation are ordered, strategy cases can be automatically derived out of the diagnostic cases. Thus, only diagnostic cases must be acquired. E.g., the following four strategy cases can be derived from the above mentioned diagnostic case:

----------------	symptom1 = 1	symptom1 = 1	symptom1 = 1
symptom1	-----------------	symptom2 = 0	symptom2 = 0
	symptom2	-----------------	symptom3 = 2
		symptom3	-----------------
			symptom4

Cases are organized in a hierarchical case memory which is an integration of the experience graph of the PATDEX system (see figure 5) and of an adaptation and specialization of the conceptual memory described in [12]. While there exists only one global case memory for diagnostic cases, strategy case memories can be considered as locally defined for each context. They support the generation of ordering rules for the respective context. The diagnostic case memory is the starting-point for the generation of shortcut rules for all contexts of the kernel system. As the direct interpretation of case memories is beyond the scope of this paper, we must refer to [9] where the fundamental procedure has been described in detail. We now focus on the GenRule system and the kernel system´s capabilities of handling uncertain information like partial shortcut rules.

7.3. Knowledge Base Refinement Using an Explanation-Based Learning Technique

One important task of the case-based component is the generation of (partial) shortcut rules by the GenRule system. Two basic procedures exist for doing this: the first one has been published in [13] and compares diagnostic cases, which have the same diagnosis, with one another. The second one has been published in [11]. It is more efficient than the first one because cases are compared to minimal diagnosis paths that have been reconstructed out of the MOLTKE knowledge base. It uses a hierarchical case memory of diagnostic cases as starting-point. From an explanation-based learning point of view (see figure 4) the MOLTKE 3.0 knowledge base is the underlying domain theory, the goal concept is to improve the diagnosis, i.e. having to ask for as few symptom values as possible,

1 All symptoms which are not mentioned in the case description are assumed to have the value *unknown.*

the examples are the cases and the operationality criterion is the shortened list of symptoms for which still has to be asked, i.e. the removal of possible tests is the smallest grain size of improvement that is wanted. If a case is presented it is compared to the reconstructed diagnosis path which has the same diagnosis. If the case is “shorter” than the diagnosis path, i.e. it is a positive example in the sense of the goal concept, then GenRule creates a new shortcut rule and adds it to the shortcut rule base of the respective context (if the determination factor exceeds a certain threshold). Thus, the improved domain theory is the starting-point for the next case that is to be processed. Therefore the described procedure is a closed-loop learning mechanism. In the case of a negative example, i.e. the case is not shorter then the diagnosis path, the procedure does nothing. The formal definitions, the description of the underlying algorithm, the computation of the determination (or certainty) factor, and the categorization of the shortcut rules according to their attached determination factors are given in [11].

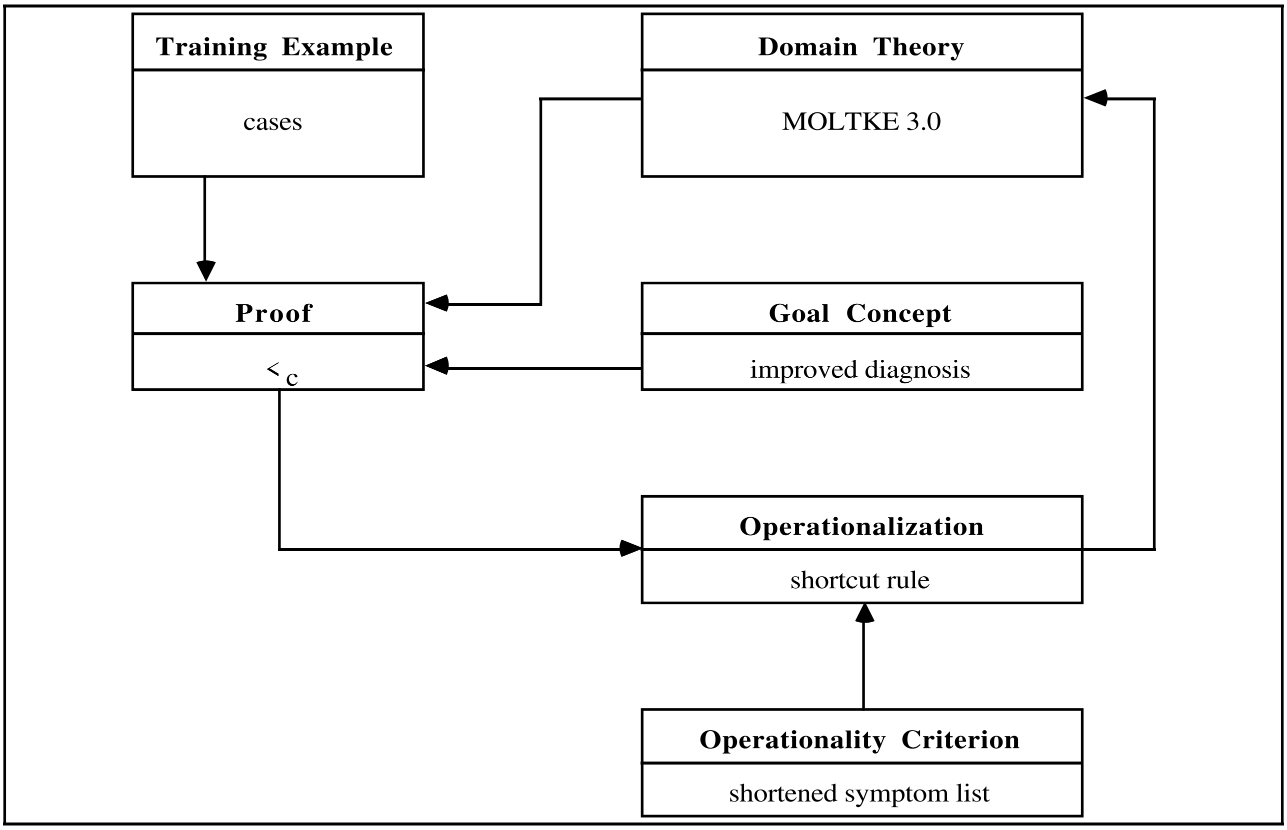

Figure 4: Adapting explanation-based learning for MOLTKE

7.4. Extracting Ordering Information from Strategy Cases

Based on the strategy case memories of the respective context the case-based component is able to generate ordering rules for the kernel system. This results in an explicit representation of ordering information which is snapshot-like. An alternative approach is the direct interpretation of the strategy case memory which enables the application of an incremental and closed-loop learning technique. Within this paper we will concentrate on the first approach.

For the extraction of ordering information from the case memory it is sufficient to describe the case memory as in figure 5. Empirical knowledge is represented by means of a weighted directed graph. While the nodes in this graph represent situations, the weights of the directed edges between these nodes represent the conditional probability of one situation (represented by the end node of the specific arc) occurring next in the diagnostic process under the assumption that another situation (represented by the start node of the arc) describes the current one[1]. Every time the strategy case memory is asked for a new symptom, which should be ascertained, the statistical information represented in the network is used to find out the (more comprehensive) situation which, by prior experience, suggests the best suited symptom to ascertain. This task is accomplished by running a heuristic-driven search through the graph. The result of the search process will be a situation and the additional symptom within this situation will be the one looked for.

The GenRule gives an empty situation to the strategy case memory which answers with a symptom that should be tested. Depending on the answered symptoms GenRule then systematically creates all possible situations which are all given to the case memory. This loop stops when all leaf nodes of the most probable paths are reached. The ordering rules are then built by the association of a situation, given as input to the strategy case memory, with the symptom that is answered by the memory.

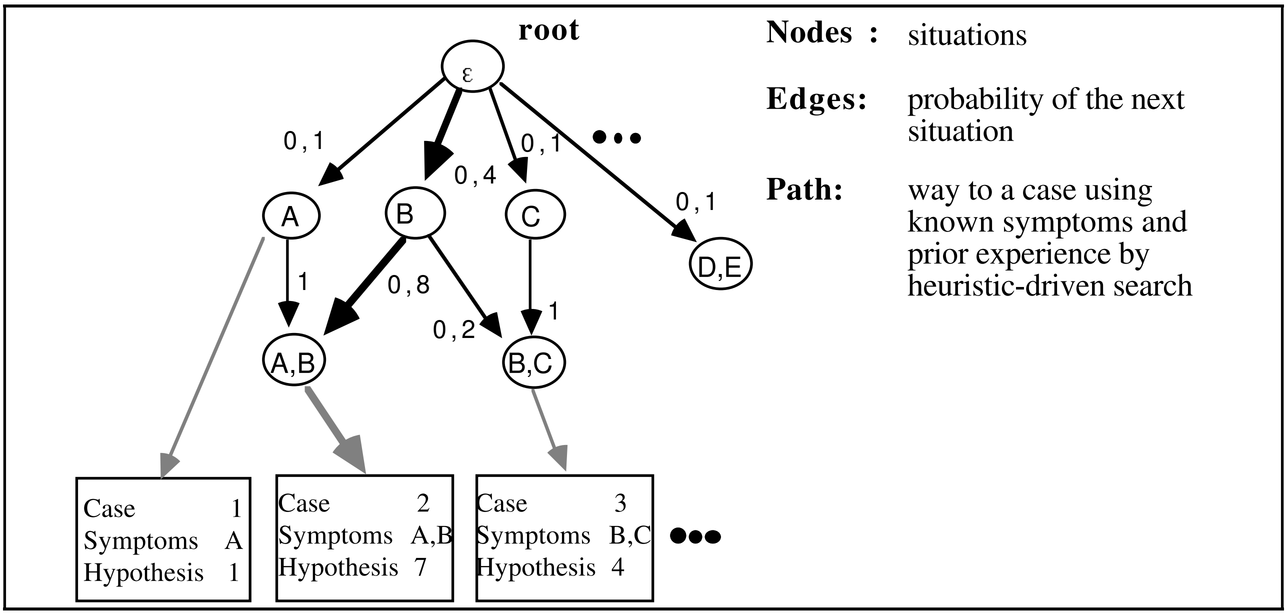

Figure 5: Representation of empirical knowledge

7.5. Utilization of Empirical Knowledge in the Kernel System

A basic requirement from a real world application in the domain of technical diagnosis (as in many others) is to make the problem solving process as transparent as possible. One of its consequences is to treat uncertainty in a rather simple manner. Within the kernel system only the partial shortcut rules are uncertain. This uncertainty is described by a list of three determination factors. All factors are computed using the arithmetic expression given in [11] but are related to different sets of cases of the diagnostic case memory. The third value depends on the set of all cases and is therefore the most conservative estimation. The second value depends on the set of cases which all have the same diagnosis and have been responsible for the generation of the considered shortcut rule. This estimation is the most optimistic one. Both values are computed by GenRule. The first value is dynamically computed by the kernel system after a diagnosis has been made. This value depends on the set of all the cases that have the same diagnosis as the actual one, unified with the set of cases

[1] The role of the weights is similar to that of certainty factors for probabilities where the underlying distribution function is not known, too.

that is the basis for the second value. If the actual diagnosis is identical to the one mentioned in connection with the second factor the first and the second factor are equal.

MOLTKE´s heuristic-based treatment of uncertainty encompasses the following:

- the kernel system does not compute any probability value during a diagnosis session
- the user can specify:
  - the worst probability category of shortcut rules that is still accepted by the user
  - how many partial shortcut rules are allowed to fire during one diagnosis session
  - how many uncertain symptom values are allowed to be used for getting shortcut rules (including the total ones) fired. This is tested by the shortcut rule interpreter.

If a diagnosis has been made all ascertained symptoms, especially all uncertain ones, are shown to the user. For the uncertain symptoms the corresponding shortcut rules, including their (first) determination factors, are mentioned. If the user states that the presented diagnosis is wrong the system makes several suggestions what to do then:

- change of (arbitrary) selected symptom values
- verification of special uncertain symptom values
- another interpretation of the knowledge base without usage of uncertain information
- another interpretation of the knowledge base with corrected symptom values
- another interpretation of the knowledge base with additional symptom values

As the kernel systems maintains a Rete-like formula network all these actions can be carried out easily. During runtime the system has to remember all uncertain symptoms and to count the number of evaluations of partial shortcut rules and the number of evaluations of shortcut rules with uncertain symptoms. The uncertainty of a symptom affects only the shortcut rule interpreter.

Besides the determination of symptom values partial (and also the total) shortcut rules can be used to define a special ordering strategy, namely to try to ascertain symptoms in a manner that maximizes the number of shortcut rules that can fire.

8. The Integration of the Multiple Knowledge Sources

In project MOLTKE two ways of integrating deep and case knowledge into the kernel system are realized. First, compilers can generate the objects used by the shell: this is done in an offline process by MAKE and GenRule (see sections above). For the shell the resulting system is not different from a user defined one. Thus, the whole knowledge acquisition environment of the shell can be used for editing the automatically generated knowledge base. This allows to combine deep modelling with case-based reasoning and common knowledge acquisition techniques.

Second, case and deep knowledge can be used online by different interpreters. We will describe this feature now in more detail.

The default interpreter uses ordering rules as compiled knowledge about strategies. Every context contains an interpreter which evaluates the ordering rule set according to the actual situation. Replacing this context interpreter by a case- or model-based one is easy because of the modularisation of the system. The new interpreter gets the actual situation as input and has to return the symptom instance to test. Thus, the main work is not the integration of the new interpreter into the kernel system but the implementation of the new facility itself.

As described above, we implemented, according to the given specification, an interpreter for model-based ordering knowledge. One possible online interpretation of the strategy case base has been described as the underlying mechanism for the generation of ordering rules. The appropriate modul can easily replace the default context interpreter. The integration of a neural network which was trained with a set of strategy cases is completed. This network gets the actual situation as input and returns the symptom to test. A detailed description of this approach is not within the scope of this paper.

As the interpretation of ordering knowledge depends only on the context interpreter of a particular context, the system is able to combine the strategy interpreters, i.e. in context A the neural network is used whereas in context B the model-based approach determines the strategy. The use of more than one technique inside a single context would be easy to implement but does not seem to be promising to us.

Besides the described two levels of integration the case- and the model-based component can interact with each other. The case-based component can ask the model-based one for a plausibility check of its cases whereas the model-based part can retrieve statistic information from the case memory.

9. Discussion

The starting point for the MOLTKE project has been the end of 1986. It has been a cooperation between our expert system research group at Kaiserslautern (Prof. Dr. M.M. Richter) and the "Laboratorium für Werkzeugmaschinen und Betriebslehre" (WZL) of the Technical University of Aachen (Prof. Dr.-Ing. T. Pfeifer), a world-wide distinguished mechanical engineering institute. The aim of project X6 was to improve the expert system technology for the diagnosis of technical systems. Therefore a complex real world domain (CNC machining centers) has been chosen to allow a deep analysis of the requirements which arise from such a domain and to enable us to develop the proper solutions. Important requirements are: representation and processing of a huge amount of knowledge, sufficient overall performance, ease of adaptability to new sets and types of the technical system, transparency of the diagnostic process, representation and processing of temporally distributed symptoms, flexibility of the diagnostic procedure and automation of the diagnostic process (as far as possible). These requirements are the generalized result of the analysis of the cooperating mechanical engineering research institute. The results have been obtained by a rather complex process and have not been clear in their whole breadth and depth at the beginning of the project. Therefore many different prototypes and specialized architectures (up to now: 10) have been implemented to enable experimental evaluation by the engineering institute.

A comparison of these architectures and the evaluation which of the just mentioned requirements are met by the MOLTKE toolbox (and how far) is beyond the scope of this paper[1]. Anyway, expert system evaluation is a hot research topic and there are no commonly accepted evaluation schemes up to now, especially not for integrated toolboxes. However, we will present an overview of our internal evaluation scheme:

- *A comparison of manually and automatically developed knowledge bases*

  This is currently under develoment for the domain of CNC machining centers.
- *An analysis of the transferability to other machines of the same domain (of CNC machines)*

  In cooperation with a mechanical engineering research institute at the University of Kaiserslautern we have successfully implemented a knowledge base for a 3D-CNC measuring machine.
- *An analysis of the transferability to similar domains*

  At the moment we are developing a knowledge base for the diagnosis of driving machines in mining.

---

[1] The basic assumptions and hypotheses, an overview of the domain of CNC machining centers, the arising requirements and necessary solutions, the main results and an overview of the implemented architectures for technical diagnosis will be published in a book, probably at the end of 1990

- *An analysis of the transferability to other technical domains*

  As an example for a quite different technical domain a diagnostic knowledge base for heterogenous computer networks is currently developed.
- *Adaptation of knowledge to new series and types of the same machine*

  This is a special research objective within our project [5].
- *A comparison of the model-based and case-based component with other approaches*

  This has partially been done up to now (e.g. for the case-based reasoner of the toolbox [9]; for the model-based component [6]), but the completion is forthcoming.
- *A comparison of alternative mechanisms within the toolbox*

  In this point we hope to get a deeper insight when all the above mentioned applications are completed.
- *An overall evaluation of the complete MOLTKE toolbox*

  We believe that the successful integration of methods from several complicated areas into one toolbox is the real research progress we can offer. The ongoing applications will help us to show the usefulness of MOLTKE and provide us with further realistic evaluation.

10. State of Realization

The kernel MOLTKE 3.0 is fully implemented including MAKE and GenRule. A case-based reasoner (PATDEX) was implemented as a stand-alone system. The integration into MOLTKE 3.0 is currently in work.

Using the MOLTKE toolbox we implemented an expert system for fault diagnosis of CNC machining centers (see [11,14]) and one for the diagnosis of 3D-CNC measuring machine [15]. Currently expert systems for the diagnosis of failures in heterogenous computer networks and for the diagnosis of driving machines in mining are developed.

The MOLTKE system is implemented in Smalltalk-80 on Sun-, Apollo- and HP-Unix-workstations. It runs on all machines for which the respective virtual machine for Smalltalk-80 was implemented including DECstations, MacII, 80386-PCs etc.

11. References

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

[2] Nökel, K.: Temporal Matching: Recognizing Dynamic Situations from Discrete Measurements, in: Proc. IJCAI 1989

[3] Althoff, K.-D., De la Ossa, A., Faupel, B., Maurer, F., Nökel, K., Rehbold, R.: MOLTKE 3.0, in: Extended abstracts of the GI-workshop on knowledge-based diagnosis systems, University of Kaiserslautern, Febr. 1990

[4] Althoff, K.-D., Nökel, K., Rehbold, R., Richter, M.M.: A Sophisticated Expert System for the Diagnosis of a CNC Machining Center, Zeitschrift für Operations Research (32), 1988, pp. 251-269

[5] De la Ossa, A.: Adaptation of Knowledge to Changes in the Physical System, Technical Report, University of Kaiserslautern 1990

[6] Rehbold, R.: Model-Based Knowledge Acquisition from Structure Descriptions in a Technical Diagnosis Domain, Proc. Avignon 1989

[7] Davis, R.: Diagnostic Reasoning Based on Structure and Behavior, in: Artificial Intelligence 24 (3), 1984, pp. 347-410

[8] Althoff, K.-D., De la Ossa, A., Maurer, F., Stadler, M., Weß, S.: Adaptive Learning in the Domain of Technical Diagnosis, Proc. Workshop on Adaptive Learning, FAW Ulm, 1989

[9] Althoff, K.-D., De la Ossa, A., Maurer, F., Stadler, M., Weß, S.: Case-Based Reasoning for Real World Applications, Technical Report, University of Kaiserslautern 1990

[10] Davies, T., Russel, S.J.: A Logical Approach to Reasoning by Analogy, in: Proc. IJCAI 1987, pp. 264-270

[11] Althoff, K.-D., Faupel, B., Kockskämper, S., Traphöner, R., Wernicke, W.: Knowledge Acquisition in the Domain of CNC Machining Centers: the MOLTKE Approach, in: Proc. EKAW 1989, pp. 180-195

[12] Kolodner, J.L.: Maintaining Organization in a Dynamic Long-Term Memory, Cognitive Science (7), pp. 243-280, 1983

[13] Althoff, K.-D., Kockskämper, S., Maurer, F., Stadler, M., Weß, S.: Ein System zur fallbasierten Wissensverarbeitung in technischen Diagnosesituationen, in: Proc. Austrian Conference on Artificial Intelligence, Innsbruck, 1989, pp. 65-70

[14] Pfeifer, T. Held, H.-J., Faupel, B.: Aufbau einer Wissensbasis für Fehlerdiagnosesysteme von Bearbeitungszentren.- VDI-Z VDI-Verlag 10, 1988

[15] Droste, K., Kaul, U.: Ein System zur Diagnose einer 3D-CNC-Meßmaschine, project thesis, University of Kaiserslautern, 1990