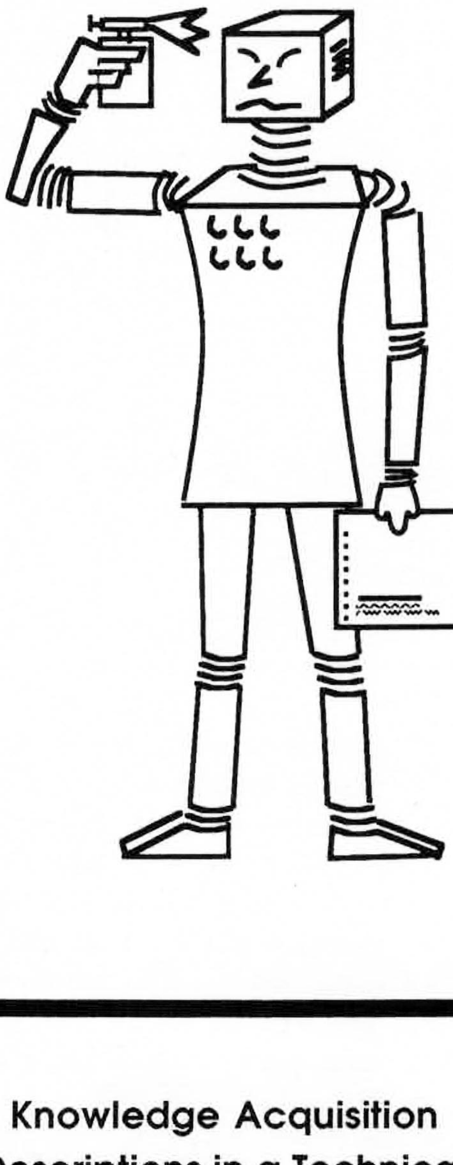
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT



Model based Knowledge Acquisition from Structure Descriptions in a Technical Diagnosis Domain

Robert Rehbold

SEKI Report SR-89-01

Model-Based Knowledge Acquisition from Structure Descriptions in a Technical Diagnosis Domain[1]

Robert Rehbold
University of Kaiserslautern
Department of Computer Science
P.O. Box 3049
D-6750 Kaiserslautern
Federal Republic of Germany
Phone +49-631-205-3357
Telex 45627 unikl d

Abstract:

To be able to quickly build expert systems for technical diagnosis causal knowledge is needed. In this paper we propose a specific way of integration between "shallow" (heuristic) knowledge and "deep" (causal) knowledge (i.e. a model of the device under investigation), namely the translation (compilation) from the model into causal rules. We present a way to separate the behavior acquisition of the typical components of a device (given by some domain experts) from the construction of a concrete representation (model) which can be built without experts. This model together with domain specific "technical common sense" integrated into our system is used to construct rules for a rule-based expert system that represent the causal knowledge . Our system is also able to modularize the resulting knowledge base into contexts (rule sets) similar to those given by experts themselves. These techniques are used in MOLTKE, an expert system for the diagnosis of CNC machining centers at the University of Kaiserslautern.

Keywords: model-based diagnosis, model-based knowledge acquisition, model compilation

1 Also appeared in the proceedings of the "Specialized Conference on Second Generation Expert Systems" at the "9th International Workshop on Expert Systems and their Applications", Avignon 1989, p. 193-205

1 Introduction

Expert systems for the diagnosis of some technical device need information about the structure, behavior and function of that device to be competent and easy to build and enlarge. This information, which we will call *causal* knowledge, differs in its sources and use from the *heuristic* and *statistic* knowledge that in the past was mainly used in expert systems. The sources for causal knowledge are - in the technical domain - quite exact, e.g. design information (function), physical laws (behavior) and connectivity descriptions (structure). All this information can be used to find faults even in new devices, i.e. no experience - inevitable for heuristic diagnosis - is needed. This is important, since diagnostic expert systems have to be made available for new devices immediately, not only after some time of experience in the field. Furthermore, new devices usually contain known parts, for which heuristic and statistic knowledge may already exist.

Most expert systems represent heuristic and statistic knowledge (also often called "shallow" knowledge) using rules. The ways in which these two can be combined with causal ("deep") reasoning still are a hot research topic. One solution to this problem is to represent causal knowledge in rules too. In this paper we propose such a concrete way of integration, the translation (compilation) of a model of a device (by model we mean a representation of the structure, behavior and function) into causal rules, i.e. rules that represent causal knowledge. We present a way to separate between the behavior acquisition of typical components of the device and the construction of the model itself, which can be built without experts from the structure description of the device. Thus we try to show how a basic rule-based expert system for the diagnosis of a device can be build from its structure descriptions and a component library. This basic expert system then contains only causal and statistic knowledge, but can be easily improved by adding other rules that represent e.g. heuristic knowledge.

The paper is organized as follows: After giving some definitions we need (section 1.1) we will take a look at the different types of knowledge and their sources (section 1.2). Then a short description of the chosen domain - CNC machining centers - is given (section 1.3), followed by an informal depiction of the task our approach tries to solve (section 1.4). The next two parts deal with the representation of structure and behavior descriptions (chapter 2) and control system error messages (chapter 3). After giving a short look at the rule system used in MOLTKE, just enough to show how our tool fits in (section 4.1) we come to the main point and show how the information from the descriptions can be used to build causal rules and to structure them (section 4.2-4.5). Some reflections on test selection and explanations together with a look at related work etc. conclude the paper.

1.1 Causal Knowledge and Deep Models

In this paper we will use the word "causal" 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 we will call a deep *model*, or just a model.

An important part of the causal knowledge is the organization of the machine which must be represented adequately in the model. Clever grouping of knowledge (of all kinds) is essential for efficient and reliable problem solving. We will call such a chunk of knowledge belonging together a *context*.

1.2 Sources of Knowledge

The different types of knowledge for technical diagnosis can be acquired from different sources:

Heuristics can only be derived from one or more experts of the domain (since this knowledge is what makes them experts) or learned from experience (as experts do).

Statistic information can be obtained from the files of the manufacturer and/or directly from the domain expert(s). If available, the former source will be more reliable than the latter due to its greater statistic significance, but may be too general to be of much use. Experts often mix statistic knowledge with other kind of heuristics to achieve their good results, often on a quite small base of typical examples.

Causal knowledge can be derived from design plans and diagrams of the device, if these descriptions are read by someone with a "technical common sense" that enables him/her to understand them. Naturally we could get the causal knowledge for our expert system from the experts too. But there is another possibility: If we provide the system with that "technical common sense" it could derive the information directly from the diagrams without using up the experts valuable time. Additionally, a systematic interpretation of the diagrams guarantees that *all* causal relations are represented in rules.

1.3 The Domain: CNC machining centers

In MOLTKE [Althoff et al. 88], an expert system project for the diagnosis of CNC machining centers at the University of Kaiserslautern, we had the problem of acquisition of causal knowledge too. In this domain even the experts (i.e. the service technicians of the manufacturer) often have to rely on the diagrams of the machine because

- there are many machine types that the same service technician has to maintain,
- nearly every other month a new series of the same machine type appears that has some changes in its composition,
- even machines from the same type and series differ due to optional subdevices.

Therefore only general statistics on typical parts are available and each individual service technician has only a low chance to acquire special heuristics for individual machine types. Thus the causal knowledge represented in the descriptions and the general understanding of the functionality of machining centers play an important role in the modeling of the diagnosticians behavior. In MOLTKE we try to integrate this description-based inference techniques with the collected heuristics of the technicians in the field.

1.4 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 that "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 in MOLTKE 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 to do the work of that engineer. The tool 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 fault identification, test suggestion and determination of yet unmeasured symptoms from this model. Furthermore the system is able to organize the rules into groups (called contexts in MOLTKE) that represent intermediate diagnoses. A more detailed description of how the rules and contexts are build can be found in chapter 4.

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 (and maybe 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. Note that our domain is not as simple as digital circuits, the usual exampĺe domain for model-based diagnosis.

2 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 (also 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 are 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 ports and states that allow to conclude 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, thus the "no-function-in-structure" principle is deliberately abandoned to enable the generation of better causal rules from the model.
- subparts and their interconnections (only if 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, which is the usual approach of model-based troubleshooting.
- 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.

We will present our notation of the model using the typical relay and switch from MOLTKE as an example[1]:

```
PrimitiveComponent define: Relay
     ports: ((currentIn medium) (currentOut medium) (lever easy))
     states: (lever (unshifted shifted) easy)
     behavior: (currentIn = +) (currentOut = -) -> (lever = shifted)
          (currentIn ≠ +) -> (lever = unshifted)
          (currentOut ≠ -) -> (lever = unshifted))
     failures: ((stuck (lever = shifted))
          (magnetDefective (lever = unshifted)))
     failProbability: high.
```

[1] Since MOLTKE is implemented in Smalltalk-80 we will use its syntax in this paper.

```
PrimitiveComponent define: Switch+[1]
    ports: ((currentIn medium) (currentOut medium) (lever easy))
    states: nil
    behavior: (lever = unshifted) -> (currentOut = 0)
        (lever = shifted) (currentIn = X) -> (currentOut = X)
    failures: (noContact (currentOut = 0))
    failProbability: medium.
```

Examples for instances are relay `s27K1` and switch `s27K1a`:

```
Relay new: s27K1
    location: toolChanger
    connections: ((currentIn m326 currentOut)
        (currentOut p1 currentIn)
        (lever s27K1a lever)
        (lever s27K1b lever)
        (lever s27K1c lever))
    position: 275.

Switch+ new: s27K1a
    location: toolChanger
    connections: ((currentIn p2 currentOut)
        (currentOut n514 currentIn1)
        (lever s27K1 lever))
    position: 214.
```

This means that a relay is a primitive (atomic) component, has three ports, two for the control current (medium measurement costs) and one for its state (low measurement cost); the lever only shifts when it gets current; typical failures are a stuck (i.e. the lever is shifted even if there is no current) and a broken magnet (i.e. the current can not shift the lever); its a priori failure probability is high, i.e. relays are a common failure reason. `Relay` only models the acting parts. Each relay can steer several instances of `Switch`, which can be an opening switch or a closing switch. Switches are connected to their relay through their `lever` port. The actual relay `s27K1` is part of the tool changer, gets its current from measuring point 326[2] and power supply 1, steers switches `s27K1a-c` and can be found on diagram position 275.

An example for a non-atomic component is the typical valve selection unit, which consists of a switch, a measuring point and a valve. This unit would be represented as follows:

```
ComplexComponent define: ValveSelection
    ports: (lever currentIn currentOut slide[3])
    states: nil
    behavior: nil
    parts: (Switch+ s MeasuringPoint m Valve v)
    connections: ((self currentIn s currentIn)
        (s currentOut m currentIn)
        (m currentOut v currentIn)
        (v currentOut self currentOut))
    failures: nil
    failProbability: unknown.

ValveSelection new: vs27Y1
    location: toolChanger
    connections: ((currentIn n152 currentOut)
        (currentOut n511 currentIn))
    subparts: ((s s27K1a 214) (m m317 214) (v v27Y1 275))
    position: nil.
```

1 `Switch+` means a closing switch (i.e. a connection is made iff the lever is shifted) while `Switch-` would denote an opening switch (i.e. connection iff the lever is unshifted).

2 Measuring points (like manometers etc.) are components whose only function is to make their input and output port the same (without failing ability). Their real purpose is to allow observations at low cost.

3 `slide` is a state of `Valve` and indicates whether the valve is open or closed.

States, behavior and failures need not be defined, instead the system will assume that the new component has the same states as its only component with state, `Valve`. The behavior of the unit will be derived from the behavior of its subparts. Failures of a non-atomic component include those of its subcomponents including the possibility of broken connections (in this case wires, which are added automatically). If no costs for tests of ports are given they are deduced from the ports the component is connected to; state test costs are inherited from the according subcomponent.

Note that the creation of instance `vs27Y1` does not include the creation of instances for `s27K1a` etc. Instead, the information about the subpart names is stored with `vs27Y1` to be used if needed. There is no need for even a single instance of `Valve` being created, if all valves are part of a valve selection component, since all rules can be derived from the class definitions.

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.

3 Control System Error Messages

The control system of the CNC machining center provides us with first information about where a failure happened by displaying an error number. These errors 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). 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 error 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.

Our tool requires the following pieces of information on each control system error message:

- error number
- textual description of failure (only needed for explanation reasons)
- name of operation(s) it can occur in
- names and values of output status when the error occurred
- names and values of input status that lead to the message

EXAMPLE:

```
ErrorMessage new: I59
     description: 'Tool cannot be released'
     operations: (releaseToolHolder)
     oStatus[2]: ((OUT7 1) (OUT24 1))
     iStatus: (((IN32 0)) ((IN36 1)))
```

[1] Ports have types and directions to ensure proper modeling (left out in this paper for the sake of readability).

[2] Input and output status are named looking from the inside of the control system. Thus OUT signals go *from* the control system to the machine; IN signals come from the machine and go *into* the control system.

4 Building Rules and Contexts from the Model

4.1 Rules in MOLTKE

Before we come to the central point of this paper, namely to show how causal rules can be derived and collected into contexts, we will take a short look at the way MOLTKE organizes its rules. As often done in rule-based systems rules are collected in groups called contexts. They were introduced to modularize the knowledge base. This modularization pays in two ways:

1) The knowledge base can be easily changed without accidentally destroying global consistency etc.
2) The speed of the system depends mainly on the number of rules in a context, not on the total number of rules in the system, thus enabling really large knowledge bases such as required by complicated devices like CNC machining centers.

Each context contains the rules to be considered in a certain state of the diagnostic process and thus represents an intermediate diagnosis. There are four types of rules in a context:

1) *diagnostic* rules to establish one or more diagnoses (intermediate or final)
2) *ordering* rules to locally order the questions in the interaction with the user (i.e. which question to ask next)
3) *determination* rules to conclude unknown symptoms out of already known ones
4) *control* rules to manage the proceeding of the system (control knowledge).

Causal knowledge is contained in the diagnostic and determination rules. The statistic knowledge incorporated in our model allows to extract ordering rules if sufficient information on the a priori failure probabilities was provided; if already known facts are considered too the border to heuristic knowledge would be crossed. Control rules contain typical expert (heuristic) knowledge and can not be derived from a model. Diagnostic and determination rules can be certain (total) or uncertain (partial). Total rules will pay for their absolute correctness[1] by having more and stronger preconditions than partial ones[2].

A rule in MOLTKE consists of the usual parts: a rule name, some conditions, some conclusions, explanations for different kinds of questions, a priority and a certainty factor (which for causal rules derived from the model is always `certain`). Conditions and conclusions can contain variables and operate on frame-based local working memories.

4.2 Building Contexts from the Model

The best way to choose contexts (remember: each context represents an intermediate diagnosis) 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. An example for a piece of a context heterarchy is given in figure 1.

The most abstract context and root of the context heterarchy is a context called `MachineFailure`. Its first level of subcontexts is induced by the control system error messages, i.e. each error number has its own context (e.g. `ErrorI59` in fig. 1). Each of these error contexts now gets subcontexts of its own for each control system input that could have been responsible for that error (e.g. `IN32=0`). Since the same fault could lead to different error messages, depending on the machine cycle it occurred in, the contexts are organized in a heterarchy instead of a hierarchy to avoid multiple instances of the same context.

[1] with respect to the underlying model

[2] Such partial rules are found in MOLTKE through case based learning.

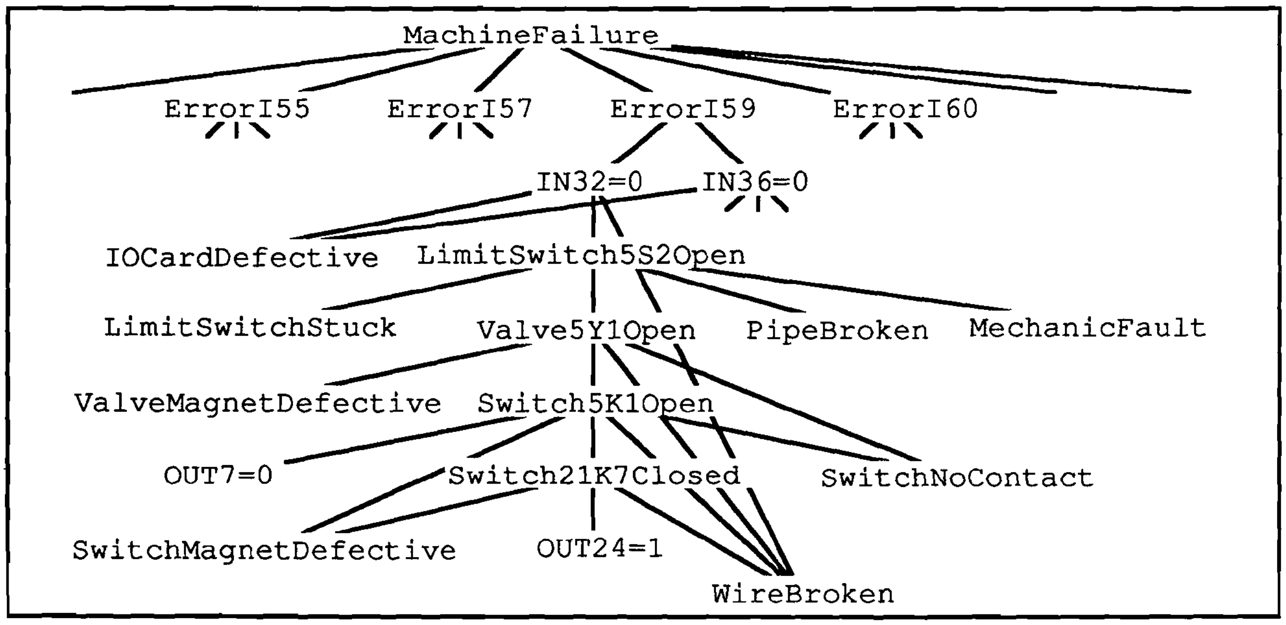


Figure 1: Example of a context heterarchy

A context C gets a new subcontext for each of its components that has a state that, if wrong (unexpected[1]), may have caused the conditions that led to the intermediate diagnosis represented by C, e.g. a valve that should have been closed was open (e.g. `Valve5Y1Open`[2]). This method is applied recursively until no more components with states are in the path from the control system output to the component with the wrong state. This path is found using the interconnection information given in the component definition. Final diagnoses (e.g. `LimitSwitchStuck`, `WireBroken`) form the leaves of the context heterarchy, since they have no need for further subcontexts. There is only one context for each of these typical faults which is instantiated with the name of the component under consideration.

Creating a context for each unexpected state of a component requires the discrimination of that component in several parts: one that represents the behavior that changes the state (which is visible as a port) and one that depends on the state in its behavior.

EXAMPLE:

> A relay is not modeled in whole, but instead separated into one "core relay" that moves the switches via the lever and the switches themselves that perform their particular function (see definition of `Relay` and `Switch` above). The port containing the effects (here: one of the switches) is treated in the context itself while the part that causes the state (here: the "core relay") belongs to the subcontext that deals with the unexpected state.

Since there may be different contexts that need a subcontext for the same unexpected state (e.g. one "core relay" can steer many switches), it is important that contexts form a heterarchy instead of a hierarchy to avoid unnecessary duplications. The separation is possible since the state itself will never be influenced by the ports it influences.

4.3 Construction of Determination Rules

A determination 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. Our determination rules rely on the fact that 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 determination rule at all. If an unmodeled fault occurs the determination rules may conclude wrong

1 We call a state wrong if it may have caused another unexpected measurement. Such states are found by tracing the effects (IN status) back to their sources (OUT status).

2 Note that there is no special context for each instance of the same class. Instead one parameterized context is created which will be used by all instances.

port values, but this kind of failure can be found by direct use of the model, e.g. by constraint suspension [Davis 84] or similar techniques which, however, will be quite costly.

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

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

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

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

EXAMPLE:

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:

Since for all X≠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 determination 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 determination rule

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

4.4 Construction of Diagnostic Rules

Thanks to the context concept the preconditions for an actual diagnosis can be local, i.e. relative to the preconditions of the whole context and that of its ancestors. Thus, a diagnostic rule consists of a number of preconditions that make sure the proper circumstances for a component to work are given, the precondition that it does not work and the conclusion to change to the corresponding subcontext, which indicates that that component is faulty, together with a high priority (since the failure is found) and certainty (since this is 100% correct with respect to the model). In the wire example above the diagnostic rule would be

```
(currentIn = X) (X ≠ 0) (currentOut = 0) -> (newContext WireBroken).
```

Diagnostic Rules lead to subcontexts, i.e. the preconditions for a diagnostic rule are the conditions under which the diagnosis (intermediate or final) of a given subcontext is established. For example, the precondition for a descent from context `ValveClosed` in our valve selection example into its subcontext `Switch+Unshifted` is `(Switch+ lever = unshifted)`; the precondition for subcontext `ValveMagnetDefective` is `(ValveCurrentIn = +) (ValveCurrentOut = -)`. These preconditions can be that (surprisingly) short, since the preconditions of the context they are used in must be fulfilled anyway and need not to be repeated. Thus, preconditions like `(OUT7 = 1)` or `(Valve slide = closed)` have not to be mentioned again.

Finding diagnostic rules from the model is quite straightforward: as mentioned above there is a context for each possible state value for each component. Its precondition is that that state value really was observed. Each failure of a component has its own context too, which is named after the failure name; the preconditions of these contexts are that their inputs were correct, while their outputs were not. Any precondition mentioned in one of the supercontexts' preconditions is simply left out.

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 determination and diagnostic 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 rules - see section 4.5 - or explanations - see chapter 5).

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

4.5 Construction of Ordering Rules

Our augmentation of the model with failure probabilities and measurement complexity allows us to produce rules that ask for easy tests that identify frequent errors first. A simple approach uses the measurement complexity first and arranges testing points of the same complexity according to the expected information, which is calculated using the model for each possible input and comparing the results to the preconditions of the diagnostic rules. A more complex solution determines the next test point from the expected increase of information [de Kleer, Williams 87]. A complete discussion of these aspects deserves a paper on its own and is omitted here.

5 Explanations

Every rule constructed from the model will be automatically augmented with a causal explanation. For the extracted determination and diagnostic rules the explanations are constructed straightforward: Every diagnostic rule gets its complete precondition (i.e. its own and all preconditions of the supercontexts it is embedded in) as its How?-, the name of its diagnosis as its Why?- and the types and names of the involved components as its What?-explanation. Determination rules rely on their associated behavior rule as their Why?- and How?-explanation and on the type and name of their component as their What?-explanation.

Even Why not?-questions can be answered due to the additional information on the possible failures, their probabilities and the measurement complexities stored with the model. This ability is deeply connected to the problem of the selection of the next test (see paragraph above), therefore we will not go into details here.

6 Further Uses of the Model

Once there is a model of the machine in the system, one could do more with it than just derive rules out of it. The model could be enhanced to be useful for all the things models are believed to do better than shallow rules, e.g. suspect generation for uncommon causes (e.g. by constraint suspension [Davis 84]), suspect examination (by simulation and testing or fault model simulation and verification) or explanation generation (by following the causal paths).

7 Related Work

[Chandrasekaran, Mittal 83] discuss the model-to-rule compilation for the medical diagnosis domain. Their main point is to show that in general a rule-based diagnosis structure D compiled from a underlying model U is able to find any disease that could be found using U. Due to their general approach they need experts to build the model U and the hierarchy of the compiled structure D.

A way of modeling technical processes is presented in [Sembugamoorthy, Chandrasekaran 86]. This approach puts the emphasis on the representation of the function of the device and assume a high-level modeling. The compilation to "specialists" seems similar to our context creation; there are no equivalents to our determination rule generation. Since the intention of their work seems to have been fairly different from ours, the high-level functional description makes it difficult to have an unexperienced user enter complex devices into the representation formalism.

[Milne 85] assigns "responsibilities" for parts of the output of analog circuits. These responsibilities are provided by an expert and use his/her terminology to reason from "second principles". The system does not compile the circuit model into rules, but instead uses a few general rules to interpret it. As [Cantone et al. 85] point out, these rules appear to be restricted to low-level components.

IN-ATE [Cantone et al. 85] deduces "missing" rules from the topology of the unit under test. The rule lists the suspect components for a given set of symptoms and distributes fault probabilities to them according to a priori failure rates it was provided with. It does not use special engineering knowledge and the rules it produces do not represent pure causal knowledge.

[Steels, van de Velde 86] propose an algorithm that learns heuristic rules from a model. The rules they derive are not certain (total), but represent a shortcut of a reasoning in the model due to omissions of intermediate steps and concentration on "plausible" conclusions. Thus, their rules are faster in finding a conclusion than ours, but do not provide the full information of the model.

[Pearce 88] uses a flat (i.e. non-hierarchical) qualitative model of a technical device to simulate the failure of each component in the model resulting in a complete example set of all failure possibilities. From theses examples his system induces rules using a version of the AQ rule induction algorithm. While the way the behavior of components is represented (using behavior rules) seems quite similar to our approach, we do not use the model for exhaustive simulation, since the size of the set of examples grows exponentially with the complexity of the device. Instead we allow a kind of restricted local simulation using determination rules.

8 Conclusion

We have presented a system to support the acquisition of causal knowledge for the diagnosis of technical devices. We have shown how a model of the concrete machine under investigation can be built from technical diagrams without the need for an engineer, and how this model can be used to construct rules for a rule-based expert system that represent the causal knowledge from the descriptions.

An expert is still needed to provide all kinds of information on the basic components, especially about failures together with their probabilities, about the difficulty of measurements and about the functionality.

A tool like ours is especially useful in a domain where similar devices are the objects of diagnosis. In the chosen domain where a new machine series with minor changes appears every other month our tool allows to derive the causal knowledge for a new series with much less effort than having an engineer doing that.

9 Status of the Research

MOLTKE is implemented in Smalltalk-80 and runs on Apollo, Hewlett-Packard and SUN workstations. It consists of a sophisticated forward/backward rule interpreter that operates on rule contexts, a frame system and several tools, e.g. a rule and a frame browser. Large parts of the acquired knowledge have been successfully integrated into the system. Due to the complexity of the machine there are still pieces of knowledge (causal and other) not integrated into the system. For the acquisition of the causal information the tool described in this paper will be quite helpful.

Up to now we concentrated our efforts on the electric and hydraulic parts of the machine for three reasons: they are simpler to model, they fail much more often than mechanical parts and they are easier to transfer from the diagrams to the system (it is possible to enter a hydraulic diagram without knowing much about hydraulics, but to model a mechanical function from technical drawings requires some skill).

10 Future Plans

More knowledge about new types and/or new series of machines will be acquired to prove the general usefulness of the presented tool.

It should be possible to automatically retrieve the error messages together with the signals that were wrong directly from the control program, which implements the machine cycles and subcycles and which gives the error messages if something goes wrong.

Mechanical components and their interactions are not really deep modeled yet. To change that is one of our main research directions and will be tackled next.

11 Acknowledgments

I would like to thank Ameen Abu-Hanna, Klaus Althoff, Klaus Nökel, Michael M. Richter and Thomas Wetter for their constructive criticism on this paper. Benedikt Faupel provided me with valuable information on the domain of CNC machining centers. Moreover I want to thank the referees for their constructive remarks that enabled me to see and (hopefully) improve on prior weaknesses of the paper.

The work presented herein was partially supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 314, project X6 MOLTKE.

12 Literature

[Althoff et al. 88]	Althoff, K., 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, p. 251-269
[Cantone et al. 85]	Cantone, R.R., Lander, W.B., Marrone, M.P., Gaynor, M.W.: Automated Knowledge Acquisition in IN-ATE™ Using Component Information and Connectivity, SIGART Newsletter No. 93, p.32-34
[Chandrasekaran, Mittal 83]	Chandrasekaran, B., Mittal, S.: Deep Versus Compiled Knowledge Approaches to Diagnostic Problem-solving, Int. J. on Man-Machine Studies (1983) **19**, p. 425-436
[Davis 84]	Davis, R.: Diagnostic Reasoning Based on Structure and Behavior, Artificial Intelligence **24** (3), 1984, p. 347-410
[de Kleer, Williams 87]	de Kleer, J., Williams B.C.: Diagnosing Multiple Faults, Artificial Intelligence **32** (1987), p. 97-130
[Milne 85]	Milne, R.: The Theory of Responsibilities, SIGART Newsletter No. 93, p. 25-29
[Pearce 88]	Pearce, D.A.: The Induction of Fault Diagnosis Systems from Qualitative Models, AAAI 88, p. 353-357
[Sembugamoorthy, Chandrasekaran 86]	Sembugamoorthy, V., Chandrasekaran, B.: Functional Representation of Devices and Compilation of Diagnostic Problem-Solving Systems, in: Kolodner, J.L., Riesbeck, C.K. (eds.): Experience, Memory , and Reasoning, Lawrence Erlbaum Assoc., 1986, p. 47-73
[Steels, van de Velde 86]	Steels, L., van de Velde, W.: Learning in Second Generation Expert Systems, in: Kowalik, J.S. (ed.): Knowledge Based Problem Solving, Prentice Hall, 1986, p. 270-295