«LE MATEMATICHE»
Vol. XLIII (1988) - Fasc. I, pp. 7-28

MODERN TRENDS IN KNOWLEDGE REPRESENTATION AND EXPERT SYSTEM TECHNOLOGY

KLAUS NÖKEL - MICHAEL M. RICHTER (Kaiserslautern)

1. Introduction.

Recent years have seen a rapid progress in expert system technology. This is due to several reasons: Development of much more powerful hardware, increased efficiency of artificial intelligence programming languages and last not least, new techniques in knowledge representation and knowledge acquisition. Besides research projects at universities and research institutions, this has led to a large variety of commercial products on the market, as well as to an increasing number of applications. We will not be concerned here with hardware aspects but we will discuss the present state of methodological concepts and software tools. Our main examples stem from the research activities as the Computer Science Department of the University of Kaiserslautern. These activities are now enlarged and complemented by the research in the

DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz) in Kaiserslautern. This is the federal institute of the FRG for artificial intelligence.

2. How to discuss expert systems and what happened in the past.

Expert systems are ultimately a program and hence a piece of software. Very often they run under the term *knowledge based systems* and it is not clear what the difference is. All the terms might also be somewhat misleading, because they suggest that one is now dealing with expert knowledge and that this was not the case in classical conventional programming. Clearly classical programs have also contained knowledge (and sometimes very sophisticated ones). Also, when an expert system is translated to machine language there is almost no way to distinguish it from a conventional program. The principle difference in this new development is a new programming style. It is the declarative programming which characterizes *AI* programming. Declarative programming means that the programmer has not to fix himself all the details of the execution of the program. Instead he puts down explicitly those facts and laws which in his opinion are necessary and useful for the solution of his problem. In order to execute the program the missing details are generated by the control structure and the inference engine. The inference engine deduces logical conclusions from the input knowledge as well as plausible hypothesis and related pieces of information. The control structure organizes this process. In the development of this programming style one can observe essentially three stages.

The first style runs under the keyword *rule based programming.* A rule is of the form

$$A_1, \ldots, A_n \to B$$

Here the A_i and the B are formulae of the first order predicate calculus. Such a rule can be used in two ways:

- forward chaining: if the system knows that $A_1, \ldots, A_n$ are true, then the truth of B can be inferred.
- backward chaining: in order to confirm the truth of B, one needs only to establish the truth of $A_1, \ldots, A_n$.

In addition to such rules one has also facts A which are considered to be unconditionally true. In the latter case variables may be involved which necessitate substitutions in order to unify the formulae in such a way that the rules can be applied as in the propositional case.

The first generation of *AI* programming languages was based on such rules. Backward chaining was realized in PROLOG and forward chaining has led to the development of OPS 5. Both languages (and their extensions) have been used to successfully implement expert systems. Although a rule is nothing but a classical IF-THEN-statement, rule-based programming may have serious advances to our traditional procedural programming. First, the development of the program may be much faster because one has to do with less implementation details. Secondly, a small change in the knowledge leads only to the replacement of a few rules and not to the rewriting of the whole program. A third aspect is that an explanation of the result is much easier than in a conventional algorithm: To know which rules have been applied very often gives sufficient information of how the result was obtained.

Rule-based programming, however, may also have serious drawbacks. If it was just considered as an advantage not to be bothered with procedural details, one might also in certain situations regret the lack of such a possibility. This is a case, for instance, if one knows an optimal algorithm and does not want to depend on the algorithm given by the inference engine. This leads to the desire to incorporate procedural elements into a declarative program. Another missing element from rule systems is the principle of modularisation and data abstraction. A third and important weakness is that there is no structure on the predicates. One cannot express that one predicate is more general than another or that some object is an instance of a general concept.

As a second style of declarative programming we mention *functional programming*. In functional programming each object is a function, and hence the distinction between a function and an argument disappears. In particular, a function can be applied to itself. A functional expression, usually consisting of a function and given arguments is converted into a certain value and the only aspect one is interested in is the relation between a functional expression and its value. This corresponds very much to the pure mathematical treatment of functions and functional programming can always be regarded as a certain kind of recursion theory or, equivalently, an implementation of the λ-calculus.

The third style is the so-called *object-oriented programming*. Objects are data capsules which have an inner life invisible to the other objects. An object has data and methods and the data belong to the inner life, i.e. they are invisible. The programming paradigm consists of a communication model; this means that objects have to send messages to each other in order to apply certain methods to certain arguments. If one has several objects with the same operations and a data space of the same structure then it is natural to form the class of such objects. It is then possible to define the common properties of those objects once and for all only for the class. Individual objects can then be considered as instances of that class. In the sense of that theory this realizes the membership relation.

The second natural step is to realize the subset relation. This leads to a hierarchy of classes which can be represented by a directed graph. An edge leads from one class to another where the edge points to a more special class 80. In fact, the above figure reflects the division of smalltalk objects into data and methods.

These programming styles were the basis for more advanced methods of knowledge representation. The most important concept in this context is that of a *frame*. A frame is a structured object which

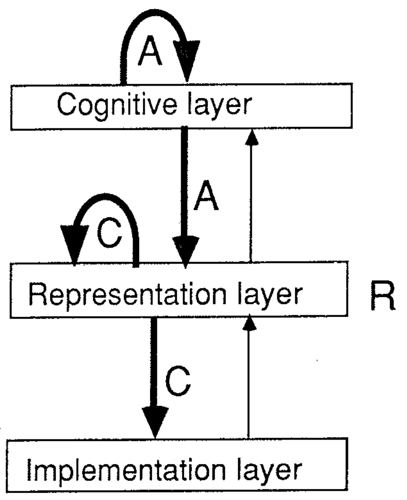

fig. 1 - Between the knowledge utterance and its machine utilization several transformations must be performed ($\Rightarrow$ - arrows). They map into the direction of increased structuring within the layers and proceed from the cognitive form to the formal, and from here to the efficiently processed form. Each syntactic result obtained in the range of a transformation must be associated with its meaning in the domain of the transformation. This is indicated by $\rightarrow$-arrows between the layers. The most interesting and difficult one is the inverse mapping into the cognitive layer, which is usually called explanation.

has some similarity with the Record of Pascal. It can be considered as a complex structured variable which has various *slots* that can be *filled* with values. The values may again be structured, in particular there may themselves be frames. Other possibilities are that slots are filled with functions, rules or simply unstructured variables. In the same way as objects are arranged by an inheritance hierarchy, this can be done with frames too. There can be however various other edges defined between frames. An example is the connected-to-edge if the frames are considered to represent components of a technical system.

It is, of course, desirable to define frames as expressive and comfortable as possible. This may however lead to serious implementation problems. Modern knowledge representation tool kits contain besides rules and functions almost always a certain concept of a frame. In most cases these systems are hybrid which means that rules, functions and frames are in different packages which are only integrated to a certain degree.

With the help of such elements of *AI* programming languages one has then built more advanced knowledge representation concepts. They aim at modelling more complex systems like complex machines. Such a machine can be considered in two ways. First, it is a static object which has parts, subparts, and so on. Secondly, the machine has a dynamic behavior which means that it can move from one admissible state to another. In order to describe such processes one needs among others an axiomatic description of time.

Using a programming language as a knowledge representation methodology one can design an expert system for a certain category in a certain domain of application. Possible categories are fault diagnosis, configuration or construction.

Such an expert system can now be considered on three levels which should be clearly separated, the cognitive level, the representation level or the implementation level.

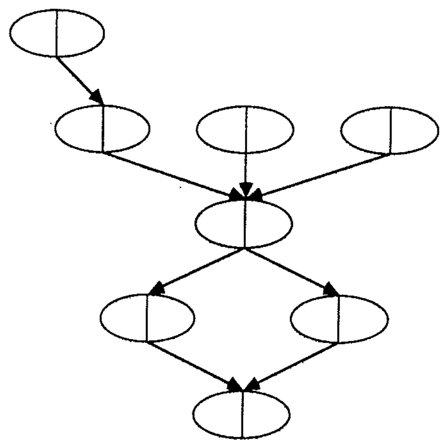

fig. 2 - Such a hierarchy allows one to store each method or operation only once in the more general object. The concept of *inheritance* means that each piece on information is transferred automatically from the superclass to a subclass if it is needed there. A programming language entirely based on the object and class principle is SMALLTALK

On the cognitive level the user wishes can be expressed. The formal representation on the second level corresponds to a specification in a conventional program. The user wishes also to describe the functional behavior, i.e. the input-output relation of the system. The functional view is realized and complemented by the architectural view which describes how the functional behavior is realized. The architectural view is realized on the implementation and partly on the representation level. In discussing an expert system it is important to point out which functional aspects lead to which architectural problems and how these problems are solved.

3. The Amalgamation Problem of Programming Languages.

Logic, functional and object-oriented programming is appropriate for the representation of different kinds of information. In reality these types of information cannot be separated well. That means, after having chosen a specific programmming language, this turns out to be suitable for some parts of intended program and less appropriate for others. This has led to the desire to integrate two or more such languages into a single one. We will discuss some of such attempts.

a) Functional and Logic Programming.

At the University of Kaiserslautern we have investigated several different approaches to the integration of functional and logic programming. In this section we describe one of these approaches, SASLOG, in greater detail.

The first decisions that have to be made in any amalgamation project concern the intensity of the integration, the functional and logic language used and the implementation technique used. Inventing a totally new language (as e.g. EQLOG [Goguen, Meseguer 86], FRESH

[Smolka 86]) forces the programmer to learn another new language and is often done to demonstrate certain abstract features clearly rather than to present an efficient implementation. For SASLOG, we instead decided to integrate two (reasonably efficiently implemented) existing languages: SASL (St. Andrews Static Language, [Turner 83]), since it features lazy evaluation and higher-order functions, and pure Prolog. Unlike many known integrations (e.g. FUNLOG [Subrahmanyam, You 86]) we really interlink these two languages in both directions, i.e. Prolog goals can be proven from SASL as well as SASL functions can be called from Prolog.

A full discussion of the two component languages is beyond the scope of this paper and can be found elsewhere. The SASL part is based on the standard set in [Turner 83] and the Prolog part is a subset of standard Prolog which leaves out all predicates with side-effects, such as e.g. *assert* and *retract*. For convenience in linking SASL to Prolog we added the object type *constant* to SASL. Constants are identifiers marked by «'» (like the abbreviation for QUOTE in Lisp); they are atomic and do *not* correspond to strings.

```
EXAMPLES:
constants:   ' john   ' mary   ' sam
def
                    lover  'mary  =' john
                    lover  x      =' sam
```

The link from SASL to Prolog is possible through two constructs: *prove* – and *ZF* – expressions.

1. Instead of any boolean expression in a SASL term there can be an expression

prove (prolog-goal)

where *prolog-goal* can contain SASL variables and expressions. If the value of the *prove*-expression is needed (remember: SASL evaluation is lazy!) the Prolog interpreter is called with the given *prolog-goal*. If the goal can be proven, the expression yields *TRUE*; if the Prolog interpreter fails the result is *FALSE*.

Regard the following SASL function which tests whether we know the mother of a given person (returning *' ok*) or not (*'unknown*):

def test x =prove(mother(_M, x)) $\rightarrow$ ' ok; ' unknown

Note that the Prolog goal contains (local) Prolog variables (_M) and parameters from the SASL function (x).

2. The more interesting link to Prolog (of which 1. is just a syntactically sugared special case) is through an extension of the ZF-expression (named after the underlying Zermelo-Fraenkel set abstraction). The syntax of a SASLOG ZF-expression is as follows:

$$[E; Q_1; \ldots; Q_n]$$

where the result term E is an expression and the qualifiers Q_k take the form

$Q_k = V_k \leftarrow E_k$ («normal» generator) or

$Q_k = [V_{k-1}, \ldots, V_{km}] \leftarrow$ *prolog-goal* (Prolog generator) or

$Q_k = E_k, E_k$ a boolean-valued expression (filter)

The meaning of this ZF-expression is very much the same as the one of $\{E|Q_1; \ldots; Q_n\}$ in mathematical notation (reading « $\leftarrow$ » for « $\in$ »), except that the ZF-expression denotes a list instead of a real set (i.e. doubles can occur and the order of members is significant).

While $V_k \leftarrow E_k$ binds V_k successively to the members of the list produced by E_k, $[V_{k1}, \ldots, V_{km}]$ $\leftarrow$*prolog-goal* binds the V_{ki} simultaneously to the values these Prolog variables take in the Prolog proof of *prolog-goal*. If the next set of values is needed, backtracking on *prolog-goal* is started.

Consider the following example, where the function f yields the list of all grandchilds of a given list of persons. Supposing the Prolog database contains several facts of the form *parent (' john, ' sue)* we can define f as follows:

def fL = [gc; old $\leftarrow L$; [gc] $\leftarrow$ parent (old, _X), parent (_X, _gc)]

If we want to filter out only those grandchilds whose parent belongs to a certain set of people we could change the definition to

def gL = [gc; old $\leftarrow L$;
    [gc, X] $\rightarrow$ parent (old, _X), parent (_X, _gc);
    member [′ sue, ′ joe, ′ john, ′ mary] X](1)

While X and gc are logic variables *inside the goal*, they become SASL parameters *outside* the generator.

Let us take a closer look at this example. Given the following database definitions:

parent (′ john, ′ sue).
parent (′ john, ′ sam).
parent (′ sam, ′ mary).
parent (′ joe, ′ linus).
parent (′ sue, ′ charly).
parent (′ mary, ′ lucy).
parent (′ jeff, ′ joe).

and the definition of g as above, then the expression $E = g$ [′ john, ′ jeff] is evaluated as follows:

a. The first element of [′ john, ′ jeff] (=′ john) is assigned to old.
b. The Prolog interpreter is started to prove parent (′ john, _X), parent (_X, _gc).
c. The interpreter returns with X bound to ′ sue and gc bound to ′charly.
d. member [′ sue, ′ joe, ′ john, ′ mary] ′ sue is evaluated to TRUE.
e. Since no more qualifiers exist this is a acceptable solution and the value bound to gc (i.e. ′ charlie) is delivered as the first element of the global expression E.

(1) Note that in SASL the order of the arguments to *member* is changed to make currying easier.

f. If the next member of E is needed, the next element from the last satisfied generator must be generated; since in this case this is the Prolog interpreter, backtracking is started.

g. Prolog finds another solution binding X to $'$ sam and gc to $'$mary.

h. member [$'$ sue, $'$ joe, $'$ john, $'$ mary] $'$ sam is FALSE so another backtracking is started.

i. Since there are no more solutions for parent ($'$ john, $\_X$), parent ($\_X, \_gc$), the Prolog interpreter fails, thus the next element from the generator of old must be examined (that is $'$ jeff).

j. A «new» Prolog interpreter is started to prove parent ($'$ jeff, $\_X$), parent ($\_X, \_gc$).

k. It finds a solution ($X =' $ joe, $gc ='$ linus) which satisfies the «member»-filter so that $'$ linus is the next member of E.

l. No more solutions for $'$ jeff can be found so the next element of L has to be used. Since there is no such element, the result list is terminated.

Thus: g['john,'jeff]=['charly, $'$ linus].

Calling SASL from Prolog is a little easier: any term in the goals on the right hand side of a Prolog clause may be an arbitrary SASL expression. These SASL terms may contain Prolog variables, which must be bound to a value when being evaluated (we do not perform residuation). Naturally SASL terms in Prolog goals can contain Prolog goals themselves (via prove- or ZF-expressions) etc. and vice versa. Note that there is no need for an operator like «is», since «=» serves the purpose equally well.

Suppose we want a predicate $P(X,Y)$ which is true, if a person X has grandchilds who are older than 20 (assuming the presence of a function age) and whose parent is either Sue, John, Joe or Mary.. It should allow to be called with any combination of bound/unbound variables. Of course we want to use our previously defined function g.

$$
\begin{aligned}
P(\_X),\_Y) :- & \text{person}(\_X),\\
& \_Y = g[\_X],\\
& \text{some}(< 20)(\text{map age}\_Y) = \text{TRUE}.
\end{aligned}
$$

```
def  some  f[]      = FALSE
     some  f[a|x]   = (fa)or (somefx)
def  map   f[]      = []
     map   f[a|x]   = [fa|mapfx]
```

Note that in a call of the Prolog predicate P the SASL function g is called which itself calls Prolog again.

One reason to integrate two existing languages instead of creating a completely new one is to give the programmer a familiar basis to work on so that programs written in either of the two languages can still be used. This puts fairly strong restrictions on the semantics of the combined language: the separate semantics have to be retained as special cases and the new elements concerning the link between the languages must be injected into their union as unobtrusively as possible. In the case of SASLOG we have chosen an operational semantics based on a version of semantic unification that is similar to the one in [Subrahmanyam, You 86].

The language was implemented in Common-Lisp on a Symbolics by Knut Hinkelmann [Hinkelmann 88] and uses the SASL interpreter written by Klaus Nökel and Robert Rehbold [Nökel, Rehbold 86] and the LISPLOG interpreter developed by Harold Boley and his collaborators [Boley 85] as essential parts.

b) Object-oriented Extension of Prolog.

Originally logic-oriented and object-oriented languages have only been loosely coupled. In the past five years several proposals for a more intensive integration of both formalism have been made (cf. [AN 86]). In principle one starts with a predicate calculus with several sorts such that these sorts are ordered by inclusion. The sorts correspond to (and are identified with) the classes in object-oriented programming. In the realization of extensions of Prolog one can basically observe three levels in the density of the integration:

- coupling of Prolog with an object-oriented language
- implementation of an object system in Prolog
- extension of Prolog by sorts.

The first approach consists mainly of the definition of a suitable interface with Prolog with e.g. Smalltalk. Using this interface Prolog programs have access to objects and their methods can again call Prolog evaluations. In principle we still have two independent processes which can call each other. A disadvantage is that these two processes both have to be active in the system. In addition, the time needed for communication is not negligible. It should be added that such a coupling exists also between the three languages Sepia Prolog, Smalltalk 80 and Common Lisp (cf. [ALT 89]); also in Babylon one finds all these three processes (cf. [PB 85]). Here the communication is organized by some kind of metaprocessor.

In order to implement an object system in Prolog one defines additional Prolog predicates, which enable the system to generate objects and to activate methods by messages. This is very similar to the flavor system in Lisp. An implementation of such a small flavor system in Prolog is described [HKPS 86].

For an object-oriented extension of Prolog the most plausible approach is to use sorts. Sorted logic can easily be implemented in Prolog. Sorts are represented by unitary predicates and inheritance corresponds to certain implications. An improvement over this simple approach uses mainly two techniques:

- Inference steps which correspond to a search in the class hierarchy are built in to the unification algorithm.
- In answer substitutions sorted variables may occur. This avoids an enumeration of all individuals via backtracking.

Such an extension of Prolog is realized on top of Prolog-XT. For more details see [ENB 89] and [MEY 89].

4. From Programming Languages to Shells.

Despite the advances in AI programming languages there is still a gap to represent knowledge in the form as an expert uses it. Data structures and interpreters for higher concepts had been developed in order to present explicitly the problem domain and the problem solving methods. This has led to the idea of a shell. Unfortunately the meaning of the term shell has changed over the years.

The first notion of a shell denotes the system which was extracted from a concrete expert system by removing all domain specific knowledge. Such a «shell» could then be used for similar problems in another domain by filling in new specific knowledge. A classical example of this type is the shell Emycin which was constructed from the medical experts with the Mycin in the way described above. Experience shows however that the domains for which such a shell could be used have to be very similar to the original domain.

Today shells denote tool kits which offer different methods like rules, functions or frames for the representation of expert knowledge. In addition shells usually contain prefabricated components, e.g. for explanation or knowledge acquisition; also help functions for a graphical user interface exist in general.

As an example we describe the *LL*-shell which was developed at the University of Kaiseslautern, see [BeSp 89].

Basic objects of the *LL*-shell are the «working memory elements» as they occur in OPS 5. Object classes describe the principle structure of all instances of such a class and their general and typical properties. Instances of a class may however very well not share these typical properties. Classes are organized in hierarchy with multiple inheritance. In this way conflicts may arise because some slot can be defined differently in several superclasses. The conflict solution consists of choosing the definition in the least abstract object which is found using the «depth-first-up-to-join-strategy».

Besides the representation of concepts as passive data elements also ideas of object-oriented programming are realized using object classes and object instances. Certain slots then correspond to sets of rules as well as goals for the backward chaining interpreter. Via the slots one has access to these active elements.

One distinguishes between methods and demons. Methods are values or slots of an instance; hence different instances of some class can have different methods. A uniform method for all instances of some class can be realized as a default value.

Demons can be associated with slots of object classes by filling the slot with the name of a rule set. If in an instance of the class the slot is accessed the forward chaining interpreter applies the rule set.

The forward chaining interpreter is organized in the classical recognize-act cycle. The partitioning into packages of rule sets acts as an modularisation technique. The backward chaining interpreter in LL-shell is a special implementation of Prolog. LL-shell has also the cut operator which however can only appear in leftmost position; this runs under the name *initial cut*.

A special feature of LL-shell is the great variety of possibilities to integrate procedural knowledge in the form of Lisp functions. One can e.g. use Lisp functions on the right side of the is-primitive in the backward chaining interpreter; hence the user can enlarge the set of standard functions by his own functions. Lisp functions can also be used in forward rules.

The acceptance of a program by potential users is essentially dominated by a good user interface. This is particularly important when one is working in expert systems. In LL-shell the design of specialized user interfaces is supported by basic functions. These include simple as well as complex routines and make an easy programming of windows and menus possible. *LL*-shell has e.g. separate functions to generate the different kinds of windows. Menus are realized as special windows. Testing and error correcting is supported by a comfortable interaction environment. This integrates different tools into a compact and easy to handle debugging component.

Moltke - an Example of a Complex Diagnostic Expert System.

There are several versions of the system *Moltke* (Models, Learning and Temporal Knowledge in an Expert system for technical diagnosis, [K.D. Althoff et al. 88]) and they have now been integrated in an overall system. The domain of application is a diagnosis of CNC-machining centers.

The first version (Fomex) is rule-based. This approach is a fault oriented one and tries to answer the following basic question: Which test must be carried out next to come to a situation in which a diagnosis can be made? In Moltke, «contexts» are the fundamental elements for enlarging the rule mechanism. They are knowledge units and facilitate combining rules in modules. The call hierarchy of the context follows the physical component hierarchy of the CNC machine. This makes possible a hierarchy of rough and intermediate diagnoses which result in the representation of faults on differnet levels of abstraction. Inside each context, the knowledge is differentiated into context and diagnosis rules. While the latter are in charge of making the diagnosis, the context rules represent explicit control information (this means they correspond to meta-rules in the usual meaning). The modular construction of the system is very clear and allows an incremental extension. Enlarging the machine with a new component requires only the creation of a new component context (and some subcontexts if necessary) and the definition of the interfaces to the other contexts via rules.

Another version (Pex) is written in a classical procedural style and can be regarded as a manually compiled form of Fomex. As expected, this version is more efficient and one could ask wether the rule-based system has been written at all. The answer is twofold and of general interest. On the one hand it would have been rather difficult and time consuming to develop Pex without having the rule-based system available and one the other hand already small changes in the knowledge base result in rather complex updates of Pex.

As a third version we mention Patdex, a pattern directed learning system. The idea is that the expert does not give a structural description of e.g. the machine but instead tells many cases from his expierence. The cases are recorded and the system compares an actual case with the case base. It selects according to a similarity measure the most similar case and uses its solution, suitably transformed, as a heuristic for a solution of the actual problem.

Finally we will comment on a special difficulty of this type of diagnostic problems and how we provided a solution.

CNC machining centers differ from many other artifacts treated in diagnostic systems (such as combinatoric electronic circuits) in that they display a complex dynamic behavior. Not surprisingly, there are also certain faults which manifest themselves in symptoms that evolve over time in a characteristic way. Often these symptoms cannot be observed by making a single measurement, instead several measurements have to be carried out in a specific order. We have studied these symptoms extensively, because they contradict several basic assumptions made in almost every diagnostic system. Most central is the broad consensus in model-based diagnosis[2] that stepwise discrimination between diagnosis candidates can be achieved by proposing a series of individual measurements and interpreting their results. However, most existing systems have gone farther than that and have made two additional assumptions:

– Each measurement result contributes *directly* to the discriminaton

[2] Rule-based systems usually make an equivalent assumption (tacitly most of the time).

process.

- Based on the measurements taken so far one can always propose a *single* successor measurement (3).

During knowledge acquisition for Moltke we first came across this type of situation, but soon we discovered more examples in every domain. Consider e.g. that you need to test whether one of the cylinders in your automobile's engine is not working properly. The usual procedure applied to each cylinder in turn would be to observe the speed of rotation with the motor running idle, and then to remove the lead from the sparking plug. A subsequent drop in the speed of rotation would indicate that the cylinder had been working alright, whereas a constant speed would imply that the cylinder had not been working all along.

In this example, three aspects are worth noting:

- The temporal order of the two measurements and the action is highly significant. This is typical of measurements which in isolation carry no information *except* through the interpretation of other measurements in their context.
- As a consequence it seems awkward to suggest the three steps of the test one at a time. Neither of the measuremets, much less the action, seem promising, if one does not have the overall effect in mind.
- The action modifies the device under consideration (and hence the model). This means that even if we were willing to stretch the meaning of «measurement» to encompass complex sequences of actions and observations, it would seem unclear how algorithms designed to find «the best location for the next measurement» could handle distributed measurements, since these cannot be assigned a meaningful location w.r.t. any one model.

(3) The claim is not that there will be one unambiguous proposal but that the planning horizon is confined to exaxtly *one* more measurement.

If temporally distributed symptoms cause so much trouble, can't we simply do without them? The answer depends on the kind of devices we are trying to diagnose. Particularly where the locations of potential measurements lie relatively dense and measurements are cheap, isolated observations are often sufficient. At the other extreme, there are cases where potentially useful measurements are in practice impossible (e.g. because they are destructive) and an observation over time may be the only way to deduce the result from more accessible sources. In between there is a wide spectrum of examples where temporally distributed symptoms are used as substitutes for isolated observations, not because the latter are infeasible in a strict sense, but simply because the former are more convenient (less costly, easier to check,...). The car engine example belongs in this category. A model-based diagnostic system that aims to choose a series of observations minimizing some measure of cost shouls reproduce this behavior of human experts.

Coping with temporally distributed manifestations of faults can be broken into two subtasks:

(1) Given two (or more) doagnoses that we would like to distinguish between, if there is no satisfying single measurement, come up with a temporal section of the device's behavior that is specific to one hypothesis and a set of actions which induce the behavior if this hypothesis is true.

(2) Given a description of this behavior, plan a sequence of interleaved actions and observations in order to detect an occurrence of the behavior. Match the actually incoming results against the description.

In Moltke we have concentrated on subtask 2, taking experts' statements of situations (=sections of behavior) as the starting point. We have designed a specification language for situations that serves as the interface between the two subtasks. Following that we have formally discussed the problem of verifying the occurrence of a situation using discrete measurements only. An algorithm that embodies a practical definition of matching has been developed and implemented. Aside from solving the problem at hand this research also provided insights into how the expressive power of Allen's interval calculus underlying the specification language can be suitably constrained in order to make it computationally more efficient. All of these results are presented in greater detail in [Nökel 88a] and [Nökel 88b].

In the future we no longer want to rely on situation descriptions given by human experts. Our current research therefore focuses on solving subtask 1. This involves primarily a partial qualitative simulation in the models corresponding to the different hypotheses (and probably in the model of the fault-free machine, too). The simulation starts at a point where two of the models diverge and tries to propagate the effects of the difference both forward to a set of observable quantities and backward to a sequence of actions that guarantees the point of divergence to be included in the behavior.

At first sight this may sound very similar to test generation techniques employed in electronics for years. However, many of the assumptions that are (legitimately) made in electronics do not carry over to our domain:

- The quantity spaces involved are mush more diverse.
- As a direct consequence the fault models can be much more complicated than simple «stuck-at» models.
- The model can change as a result of actions by the observer.
- Asynchronous processes require a more general treatment of time.

We consider each of these items an important research problem which deserves further investigation.

REFERENCES

[1] [Althoff et al. 88] Althoff K.D., Nökel K., Rehbold R., Richter M.M., *A Sophisticated Expert System for the Diagnosis of a CNC Machining Center,* in: Zeitschrift für Operations Research, Vol. 32, (1988), 251-269.

[2] [AN 86] AIt-Kaci H., Nasr R., *Login: Electric Programming Language with Built-in Inheritance,* Journal of Logic Programming 3 (1986), 185-215.

[3] [BeSp 89] Bernardi A., Spieker P., *LL-shell: Eine hybride Expertensystementwicklungsumgebung,* manuscript Kaiserslauten 1989.

[4] [Boley 85] Boley H., and the Lisplog group: *Lisplog: Momentaufnahmen einer Lisp/Prolog-Vereinheitlichung,* Memo Seki-85-03, Universität Kaiserslauten, 1985.

[5] [Clocksin, Mellish 84] Clocksin W.F., Mellish C.S., Programming in Prolog, Berlin, 1984.

[6] [END 89] Enders R., *The object-oriented extension of Prolog XT.,* Technische Berich INF 2 ASD-7-89 Siemens AG 1989.

[7] [Goguen, Meseguer 86] Goguen J.A., Meseguer J., *Eqlog: Equality, Types, and Generic Modules for Logic Programming,* D. deGroot, G. Lindstrom: Logic Programming - Functions, Relations and Equations Prentice-Hall 1986.

[8] [Hinkelmann 88] Hinkelmann K., *Saslog: Eine funktional Sprachintegration mit Lazy Evaluation und semantischer Unifikation,* SEKI Working Paper SWP-88-6, Universität Kaiserslautern, 1988.

[9] [HKPS 86] Huss K., Küchenhoff V., Pichler C., Schmauch C., *Objekt-orientierte Programmierung in Prolog.,* If Prolog newsleter 1 (3), (1986), 3-11.

[10] [MEY 89] Meyer M., *Ein Debugger für die objekt-orientierte Erweiterung von Prolog XT.* Diplomarbeit, Universität Kaiserslautern 1989.

[11] [Nökel, Rehbold 86] Nökel K., Rehbold R., *SASL: Implementierung einer rein funktionalen Sprache mit Lazy Evaluation,* SEKI Working Paper SWP-86-07, Universität Kaiserslautern, 1986.

[12] [PB 85] de Primio F., Brewka G *Babylon: Kernel System of an Integrated Environment for Expert System Development and Operation,* in: Expert Systems and their Applications, Avignon, (1985), 573-583.

[13] [Smolka 86] Smolka G., *Fresh: A Higher-Order Language Based on Unification,* in: D. de Groot G., Lindstrom., Logic Programming - Functions, Relations and Equations Prentice-Hall 1986.

[14] [Subrahmanyam, You 86] Subrahmanyam P.A., You J.H., *Funlog Computational Model Integrating Logic Programming and Functional Programming,* in: D. de Groot G. Lindstrom: Logic Programming - Functoins, Relations and Equations, Prentice-Hall 1986.

[15] [Turner 83] Turner D.A., *SASL Language Manual,* (revised version), University of Kent, Canterbury, 1983.

Univeristät Kaiserslautern
Erwin - Schrödinger - Strasse
6750 Kaiserslautern