APPLICATIONS OF ARTIFICIAL INTELLIGENCE IN ENGINEERING PROBLEMS

Proceedings of the 1st International Conference,
Southampton University, U.K.
April 1986

Volume II

Editors:

D. Sriram
R. Adey

A Computational Mechanics Publication

Springer-Verlag
Berlin Heidelberg New York Tokyo

D SRIRAM

Department of Civil Engineering
M.I.T.
Cambridge
Massachusetts 02139
USA

R ADEY

Computational Mechanics Inc
Suite 6200
400 West Cummings Park
Woburn MA 01801
USA

British Library Cataloguing in Publication Data

International Conference on the Application of Artificial Intelligence in Engineering Problems (1986: University of Southampton) Proceedings of the 1st International Conference on the Application of Artificial Intelligence in Engineering Problems, Southampton University U.K., April 1986.
1. Engineering design—Data processing
2. Artificial intelligence 3. Pattern recognition systems
I. Title II. Sriram, D. III. Adey, R.
620'.00425'0285 TA174

ISBN 0-905451-47-3

ISBN 0-905451-47-3 Computational Mechanics Centre, Southampton
ISBN 3-540-16349-2 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-16349-2 Springer-Verlag New York Heidelberg Berlin Tokyo

This Work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically those of translation, reprinting, re-use of illustrations, broadcasting, reproduction by photocopying machine or similar means, and storage in data banks. Under §54 of the German Copyright Law where copies are made for other than private use, a fee is payable to 'Verwertungsgesellschaft Wort', Munich.

© Computational Mechanics Publications 1986
Springer-Verlag Berlin, Heidelberg
Printed in Great Britain.

The use of registered names trademarks etc. in this publication does not imply, even in the absence of a specific statement that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

VMES: A Network-Based Versatile Maintenance Expert System[1]

Stuart C. Shapiro, Sargur N. Srihari, Ming-Ruey Taie, James Geller
Department of Computer Science, State University of New York at Buffalo, Buffalo,
NY 14260, USA

ABSTRACT

We are developing a versatile maintenance expert system (VMES) for trouble-shooting circuits. Like several other research teams we are using structural and functional descriptions to avoid difficulties of empirical-rule-based diagnosis systems in knowledge acquisition, diagnosis capability, and system generalization. Our diagnosis system has successfully pinpointed the faulty part of a multiplier/adder board, a favorite example for researchers in this field. We have embedded VMES in the SNePS Semantic Network Processing System, using it as a form of expert system tool. A central part of VMES is the "SENDING" graphical interface. While troubleshooting, it displays the given device and dynamically indicates the state of the reasoning process. All knowledge used by "display" is directly retrieved from the semantic network. This operation of "display" is comparable to natural-language generation from a knowledge base. An important aspect of our research is to find a good knowledge-representation scheme to support diagnosis and display.

INTRODUCTION

Diagnosis expert systems have been developed for two main areas of application, hardware trouble-shooting and medical diagnosis. Some diagnosis systems, such as MYCIN [Shortliffe76] for medical diagnosis and CRIB [Hartley84] for computer fault diagnosis, are built on rules which represent empirical associations. Though these systems have had considerable success, there are some important drawbacks: knowledge acquisition from domain experts is difficult, all possible faults (diseases) have to be enumerated explicitly which results in a limitation of the diagnosis power, and such programs have almost no capability of system generalization.

Structural and functional descriptions, usually referred to as "design models" of a device, have been widely used by fault diagnosis researchers as a solution to

[1] Work reported here was supported in part by the Air Force Systems Command, Rome Air Development Center and the Air Force Office of Scientific Research.

the difficulties of empirical-rule-based diagnosis systems in knowledge acquisition, diagnosis capability, and system generalization [Davis84, Genesereth84]. The knowledge needed for building such a system is well-structured and readily available at the time when a device is designed. There is no need to explicitly enumerate all possible faults since they are defined generically as violated expectations at the output ports. This approach makes the adaptation for the system to a new device much easier, because the only thing needed is to describe the device to the system.

For our own research, we have implemented a diagnosis system called VMES (Versatile Maintenance Expert System) that has successfully pinpointed the faulty part of a multiplier/adder board, a favorite example for researchers in this field (see e.g., [Davis84].)

Since a design-model-based fault diagnosis system reasons directly on the structure and function of a device and usually uses a simple inference engine, the knowledge representation is vital to the performance of such a system. Many researchers use predicate logic, but this has drawbacks: the representation, the resolution rule, and the diagnosis assumptions seem fairly unnatural [Fikes85]. We are implementing our system using SNePS the "Semantic Network Processing System"[Shapiro79]. SNePS provides several advantages which will be discussed in detail later in the paper.

VMES contains an integrated knowledge base and a device independent inference engine. A hierarchically arranged knowledge base provides abstraction levels of devices, and makes the inference engine able to focus on a limited number of objects at any time. Initially only component *types* are represented in the knowledge base, an object is instantiated only when needed. Since devices in the domain share common components, this approach avoids redundant representations. When the system is adapted to a new device, the only thing needed is to add descriptions of the new component *types* used by the new device.

The main mode of communication between the SNePS reasoning mechanism and the user is intended to be a graphical interface. We have implemented a version of such an interface called "SENDING" that is used in tracing the reasoning process of the Adder/Multiplier diagnosis program. Besides the user friendliness and naturalness of such an interface we want to use it as a testbed for research in the area of visual knowledge.

While visual knowledge has been dealt with implicitly in Computer Vision and from a different aspect in Computer Graphics, and in Cognitive Psychology for quite some time, we lately have been experiencing a growing interest in an explicit treatment based on Knowledge Representation methods. [Latto84, Davis85]. The crucial point here is the interest in a natural representation that lends itself to reasoning processes as opposed to a representation for ease of "recognition".

STRUCTURAL DESCRIPTION

Although we are planning to use multiple sources of knowledge for the process of diagnosis, in our current implementation, only the logical *structure* of a device is represented and used. Instead of hand-coding every detail of the device, the system keeps a component library which describes every "type" of component. Each component type is abstracted at two levels and represented by two SNePS rules which are categorized as instantiation rules. The structural representation reflects the part hierarchy of a device. Sub-parts of a device are instantiated only when they are needed. This increases memory efficiency.

At level-1 instantiation, an object is built as a module (a black box) with its I/O ports and a pointer to its functional description. The functional description is implemented as a LISP function which simulates/infers the value of one port in terms of the others. This will be discussed later.

At level-2, the sub-parts of the object at the next hierarchical level are built, and the wire connections between the object and its sub-parts, as well as those among the sub-parts themselves are created. When being built, each sub-part is assigned a name which is an extension of the name of its super-part (the object), and it is instantiated at level-1 so that its I/O ports are available for the wire connections.

The instantiation rules for objects of the type "M3A2" are as follows: (M3A2 is an artificial multiplier/adder board which consists of three multipliers and two adders. See Figure. 1 for its structure.)

All annotations are shown in italics.

level-1 description: I/O ports and functions.

```
(build
   avb $x
   ant (build object *x type M3A2 state TBI-L1)
   cq (build inport-of *x inp-id 1) = vINP1
   cq (build inport-of *x inp-id 2) = vINP2
   cq (build inport-of *x inp-id 3) = vINP3
   cq (build outport-of *x out-id 1) = vOUT1
   cq (build outport-of *x out-id 2) = vOUT2
   cq (build port *vOUT1 f-rule M3A2out1
                pn 3 p1 *vINP1 p2 *vINP2 p3 *vINP3)
   cq (build port *vOUT2 f-rule M3A2out2
                pn 3 p1 *vINP1 p2 *vINP2 p3 *vINP3]
```

The first three lines say that "if x is an M3A2 and is to be instantiated at level-1 (TBI-L1), then do what follows".
The next five lines instantiate the I/O ports. The last two "builds" link the output ports to the functional description of the object.
The first one says "in order to simulate the value of the first output, use the function M3A2out1 which takes three parameters namely the inputs of the object x in correct order".

level-2 description: sub-parts and connections.

```
(build
   avb *x
   ant (build object *x type M3A2 state TBI-L2)
   cq (build
      avb ($xp1 $xp2 $xp3 $xp4 $xp5)
      ant (build name: Give-PID-M3A2 object *x
                p1 *xp1 p2 *xp2 p3 *xp3 p4 *xp4 p5 *xp5)
      cq ((build object *xp1 type MULT state TBI-L1)
         (build object *xp2 type MULT state TBI-L1)
```

```
(build object *xp3 type MULT state TBI-L1)
(build object *xp4 type ADDER state TBI-L1)
(build object *xp5 type ADDER state TBI-L1)
(build super-part *x
     sub-parts (*xp1 *xp2 *xp3 *xp4 *xp5))
(build from *vINP1
     to  ((build inport-of *xp1 inp-id 2)
          (build inport-of *xp2 inp-id 1)))
; to save space, not all wire connections are shown here.
(build from (build outport-of *xp3 out-id 1)
     to  (build inport-of *xp5 inp-id 2))
(build from (build outport-of *xp5 out-id 1)
     to  *vOUT2]
```

The first seven lines say: "if x is an M3A2 at level 2 (TBI-L2), use function Give-PID-M3A2 to generate names for x's sub-parts".
The next seven lines declare the types of the sub-parts which will activate appropriate rules to instantiate them at their level-1.
The super-part/sub-parts hierarchical relation between the object x and its sub-parts is also built.
The remainder of the example connects the wires between x and its sub-parts as well as those among the sub-parts themselves.

All instantiation rules are stored in a file, which is regarded as a component library. Representing the structure of a device via the instantiation rules and the use of a component library gives the system several important advantages. We do not have to hand-code knowledge representations for several almost identical elements on a digital circuit board; also memory is saved, because the representation is generated by the system only when required during the course of diagnosis. In other words we gain efficiency at the system development time, as well as during diagnosis. We have found that this is especially important in a memory critical environment.

Although instantiation during diagnosis is good for memory efficiency, it is slower during diagnosis because of the time required for instantiation. To overcome this problem without degrading the benefit of fast system construction, we designed the representation in a way which allows pre-instantiation of the device before diagnosis. This can be done easily by changing all TBI-L2 nodes in the component library to TBI-L1. Since the instantiation rules are used in a forward way, if a device is declared to be some type at its level-1 instantiation, it would activate all required instantiation rules throughout its structural hierarchies and build every detail of the device. This design gives the system one more dimension of versatility, namely that the system is versatile in both memory-critical and diagnosis-speed-critical situations.

The most important advantage of the current implementation is the extreme ease in adapting the system to other devices. All that the user has to do is to add the structural and functional information of the "new" component types to the component library and the function library, which will be discussed later. A new component type is defined as a component type which has not been described to the component library. The new device itself is a new component type by our definition. The effort required to adapt the system to new devices should be minimal since digital circuit devices have a lot of common components, and the structural and functional description are readily available at the time when a device is designed.

FUNCTIONAL DESCRIPTION

The functional description should be useable to simulate the component behavior, i.e., to calculate the values of output ports if the values of the input ports are given. It should also be useable to infer the values of the input ports in terms of the values of other I/O ports. This is important if hypothetical reasoning is used for fault diagnosis. Though we have only used the functional description to calculate the value at the output port, our representation scheme can be used both ways.

The functional description is implemented as a LISP function, which calculates the desired port value in terms of the values of other ports. Every port of a component type has such a function associated with it, the link between the port and the function is described in the structural description. Since different ports of different component types might have the same function, some functions can be shared. Several examples of functional descriptions follow:

All annotations are shown in italics.

Below is the function for the first output port of M3A2-type objects

```
(defun M3A2out1 (inp1 inp2 inp3)
    (plus (product inp1 inp2)
          (product inp1 inp3)))
```

Below is the function for the output port of MULTiplier-type objects

```
(defun MULTout1 (inp1 inp2)
    (product inp1 inp2))
```

Below is an artificial example to show a function shared by several different component types namely by the type "super-buffer", the type "wire"and the type "1-to-1 transformer".
All these component types show the same behavior at our level of component abstraction: they echo the input to the output.

```
(defun ECHO (inp1)
    inp1)
```

As depicted above, the functional description is versatile in that it supports the simulation and the inference of the device behavior; it also supports hypothetical reasoning and the representation scheme is quite simple.

INFERENCE ENGINE

The inference engine for fault diagnosis follows a simple control structure. It starts from the top level of the structural hierarchy of the device and tries to find output ports that violate an expectation. "Violated expectation" is defined as a mismatch between the expected (calculated) value and the observed (measured) value at some output. After detecting a violated expectation, the system uses the structural description to find a subset of components at the next lower hierarchical level which might be responsible for the bad outputs. This process is then continued with the suspicious parts. A part is declared faulty if it shows some violated expectation at its output port and it is at the bottom level of the structural hierarchy. The bottom of the hierarchy will contain the smallest replaceable units for the intended maintenance level. In other words, if a device can be replaced but not repaired in a certain situation, then there is no need to represent its internal structure.

The inference engine is a rule-based system implemented in the SNePS Semantic Network Processing System. The control flow is enforced by a LISP driving function called "diagnose". SNePS can do both forward and backward inference, and it is capable of doing its own reasoning to diagnose a fault. The LISP driving function has been introduced for efficiency reasons only.

A small set of SNePS rules is activated at every stage of the diagnosis. For example, three rules are activated when reasoning about a possible violated expectation of a specific port of a device. One rule is to deduce the measured value of the port. If the value can not be deduced from the wire connections, the rule would activate a LISP function which asks the user to supply one. A similar rule is activated for the calculated value, and the last rule is used to compare the two values to decide if there is a violated expectation. The last rule is shown below in both SNePS code and in English.

In SNePS code:

```
(build
    avb ($p $vc $vm)
    &ant ((build port *p value *vc source calculated)
          (build port *p value *vm source measured))
    cq (build
          min 1 max 1
          arg (build name: THEY-MATCH p1 *vc p2 *vm)
          arg (build port *p state vio-expct]
```

In English:

If the calculated and measured values of port p are known as vc & vm, one and only one of the following statements is true:
(1) vc and vm agree;
(2) port p displays a violated expectation.

The diagnosis strategy along with the combination of a LISP driving function and SNePS rules turns out to be very efficient. The diagnosis can be monitored by the SNePS text or graphic inference trace.

THE GRAPHICAL INTERFACE

Motivation

An important part of VMES is its graphical interface which comprises a separate subsystem called "SENDING" (SEmantic Network Domain Interface Graphics). Our interst in this interface is twofold. Currently there is a growing interest in multi media communication. Looking at technical literature which would be impossible without diagrams and drawings it becomes immediately clear that a dialogue about a technical object like a circuit board would profit very much from a graphical component. This component can be used by both the user and the computer to refer to parts which are currently under discussion.

The second source of our interest in graphical interfaces is of a more theoretical nature. We are investigating principles of visual knowledge representation. In computer vision or computer graphics, representations are mainly designed in order to permit efficient recognition or projection of objects. We are interested in representations that can be used in reasoning, as well as for display purposes.

Components of the Interface

The SENDING graphical interface contains several parts, the most important of which are the "display" function and the "readform" function. The readform function is our (simple) version of a CAD device. It permits a user to create a simple object, consisting of arcs, lines, circles, boxes, text, etc. by drawing them on the screen of a graphics terminal. Objects can contain several unconnected parts and are stored immediately as a named objects, namely as LISP functions.

Even more important than readform is the "display" function. Display takes one or more nodes of a semantic network as arguments. These nodes can be either base nodes, representing objects, or assertion nodes, representing simple propositions about one object. Assuming the semantic network contains propositions about form, position and attributes of an object, "display" can retrieve this information and create a picture of the object on the screen.

It should be noted that this approach to image generation is different from the techniques usually employed by computer graphics programs. Our object descriptions are given in a declarative format, incorporating them together with a part and a type hierarchy into a single network.

How display works

The "form" itself is a LISP function (created by readform), which is represented in the semantic network as a base node whose node label is identical to the function name. This method of picture generation is comparable to a language generation program that takes a semantic network as its input and generates a surface utterance from it.

The detailed process of displaying an object is: first the part hierarchy is used to retrieve subparts of the given object; then forms and positions of all parts are retrieved. We are permitting several different methods of positioning which are expressed with different case frames in the network. The simplest case is absolute positioning in device coordinates. More involved are relative position of an object to another object or to its super-object. The most complicated version retrieves the relative position of a part relative to its super-part by using the type hierarchy that part and super-part belong to.

After knowing position and form, attributes of objects are retrieved. Attributes can be either symbolic attributes or iconic attributes. An iconic attribute is directly displayable, and the simplest form of such an attribute is "color". Symbolic attributes have to be mapped into iconic attributes, in order to make them displayable. For instance we are marking faulty objects by changing their genuine color into a signal color (red). In this case the same medium (color) is used to express a different fact.

Attributes in our system are teated in a way that we have not seen described in the literature before, namely by making the attribute class itself a LISP function. An attribute value is passed to this LISP function as an argument (sometimes a dummy value), together with the form function, effectively making the attribute-class function a functional. The returned value of the attribute-class function is again a form function, but it is modified according to the given attribute.

Our approach to attributes guarantees that we can apply new predicates to old forms, without ever changing the form-functions. Any alternative that comes to mind would require adding new parameters to form-functions.

Special display Parameters

Modality The display function permits the user to specify a number of different parameters. One is a "modality" parameter. In our maintenance domain we are dealing with structural and functional properties of objects. This implies that it is possible and desirable to display objects in both these aspects (or as we say, modalities). The user can select which of the stored aspects he wants to see, by specifying the modality parameter accordingly. Functional display is the default.

The modality parameter is perfectly general and can be extended to any number of different aspects, however we currently see no need for others than structural and functional displays. Assertions for different modalities are not structured in a Hendrix type [Hendrix79] partition system but they contain a modality slot in the object description case frame.

Our current research has led us to the result that structural and functional displays should be treated differently, and we will talk about this more in the section on future work.

Pruning the display If a display function is used as an intelligent system as opposed to a simple mapping from a data structure to a display device, there has to be a way to "prune" the display to avoid "overloading" the user, by presenting irrelevant and therefore confusing information. One of our goals in this project is to find a method to create a cognitively appealing representation.

Several optional parameters for display have been defined, that permit the user to control the amount of information that he receives. Our goal is to automatize this process entirely, but currently the user has to decide himself what he considers appealing. The following paragraphs contain a description of these user options.

As mentioned before, our representation uses a part hierarchy. A "level" parameter permits the user to limit the number of levels in the part hierarchy that are displayed. If, for example, an object has sub-parts which have sub-parts in turn, it is possible to limit the display to showing only sub-parts, but not their sub-parts (i.e. the the sub-sub-parts of the object are not shown). Any number of levels can be represented in the semantic network, and correspondingly any

natural number can be specified for the level parameter.

Sometimes the number of effectively *visible* objects might be responsible for overloading of the user. Therefore an "objects" parameter limits the number of (sub)objects displayed. As in the level case, objects are retrieved from the part hierarchy by using breadth first selection. If the specified number of objects has been shown, display will terminate in the midst of a level.

In our current representation there is no way to express different importance for different sub-parts; therefore an "object" parameter results sometimes in displaying "unimportant" parts, a problem which has been criticized by several users. We plan to investigate this question in the future.

Objects in the VMES system can themselves be of quite different complexity. A simple wire is an object, but a 16 leg integrated circuit is also one object. In order to take care of this problem another display option has been programmed, the "complexity" parameter.

Display's "complexity" parameter extends the ideas developed above by counting not the number of objects, but the number of graphical primitives contained in them. So it is possible for the user to limit the number of *graphics primitives* that are displayed. In this way two display calls with the same "complexity" parameter might create either a picture of a simple object with five sub-parts, or a picture of a complicated object with only one sub-part.

We are still not satisfied with this solution, because there are graphics primitives of different complexities. A polygon is definitely more complex than a circle or a point. However we have not yet implemeneted a way of grading complexities of graphics primitives. We suspect that Gestalt theory could supply a theoretical foundation for such an analysis.

Optimal screen use Another type of display option deals with the use of the given screen space, the so called "fill" option. If display is called with the "fill" option, it dynamically computes its own window to viewport mapping to guarantee an optimal use of the given (globally specified) viewport. This option is also the only way to display parts of the world that do not fit into screen coordinates. In this way a user sees small objects at a reasonable size, while large objects still fit into the screen. Still he does not have to know anything about viewports and windows.

An extension of the "fill" option is the "intell" option. It constitutes another step in giving the system possibility to decide what to display. Although the name "intell" seems a little bit ambitious at the current moment, it is definitely a step towards having the system figure out what the user wants to see. The intell option is the solution for the following problem. If a user requests to see a certain object, he might at the same time be interested to see where this object "fits in" (he might not know that).

A user might also want to know if there are several other objects of the same type. If display is called with the "intell" option it will display the user specified object(s) in one viewport and in another viewport, will show the chain of all super-objects of the user specified object(s). Currently the default viewports are the left half of the screen for the object, and the right half of the screen for the super-objects. Every super-object will be shown to two levels depth (see "levels" above). So if a user displays a leg of an AND gate, then the AND gate with all its legs will be displayed. If the super-object of the AND gate is a board, then the board will be displayed with all its gates, but not with their legs. The use of the "intell" option is shown in Figure 1. Figure 1 was created with a printer that directly dumps a screenfull from a graphics terminal.

Graphical Inference Trace

The SNePS system has a tracing facility which permits a user to watch the reasoning process of SNePS. The function that is used for tracing is independent of SNePS, and it is possible to plug different interfaces into this position. An important aspect of display is that it can be used as such an interface. In other words, an observer can watch what SNePS is currently "thinking" about.

In our implementation of a diagnosis system for the Adder/Multiplier board that we have mentioned above, the system marks parts that it is currently "thinking" about by displaying a questionmark above them, and parts that it found a conclusion about by showing an exclamation mark above them. The faulty part is shown in the final display in red.

This is a direct consequence of SNePS figuring out that the part is bad. Using the attribute mechanism described above, the "state" attribute class is automatically translated into the signal color red. After the reasoning process has terminated, any display command of the object found faulty will again be in the new color. This is the case, because the semantic network has been changed permanently by the reasoning process.

DISCUSSION

An important aspect of our research is to find a good knowledge-representation scheme to support diagnosis. Many researchers use standard predicate logic, but this has several drawbacks: the representation, the resolution technique, and the diagnosis assumptions seem fairly unnatural [Fikes85]. We have implemented our

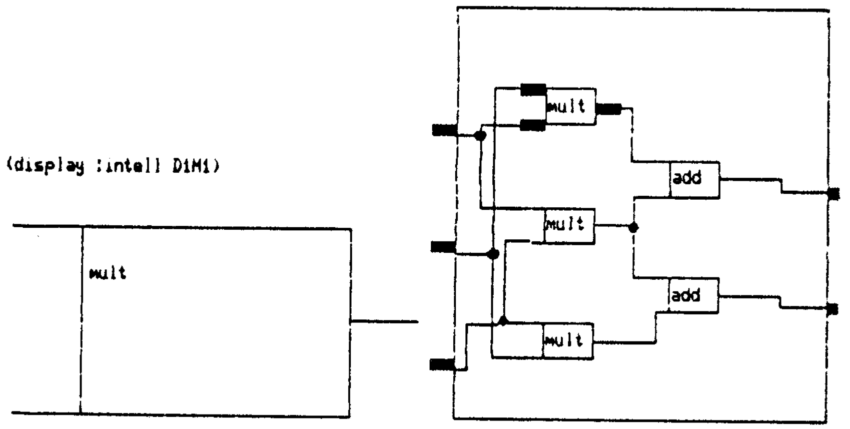
```
(display :intell D1M1)
```

Figure 1: A Multiplier and its Board

system in the SNePS Semantic Network Processing System [Shapiro79]. Advantages are: (1) structural and functional knowledge is integrated into a single network; (2) the powerful SNePS non-standard connectives permit us to express rules of a degree of complexity which most Horn clause based systems cannot use; (3) diagnosis assumptions are handled in a natural way; (4) the deduction process can be monitored; (5) inference can be traced graphically; (6) the representation can be easily expanded and modified; (7) procedural knowledge is represented and used; (8) it is smoothly interfaced with LISP.

The structural description is represented by instantiation rules at two different levels. This scheme turns out to be very effective and flexible. It can be used to pre-instantiate the target device with only little change. We ran the system on the same M3A2 board in both regular mode, which did the instantiation only when needed, and pre-instantiation mode. As expected, the former proved to be relatively more memory efficient, the latter better concerning diagnosis speed.

The main feature of our device representation scheme is the versatility of the system. To adapt the system to new devices, the only thing that needs to be done is to add new components to the system's libraries. In order to test this idea as well as the suitability of hierarchical structural representation, we invented another artificial device type called XM3A2 and entered its description into the system. The XM3A2 type has three inputs and two outputs exactly like the M3A2 type, but it only has a single sub-part which is of M3A2 type. Actually, it is a device which has an extra layer of packaging on top of an M3A2 type device.

Given that the M3A2 type has been known to the system, only the XM3A2 type had to be added, which was done by adding two simple instantiation rules. There was no need for a new functional description since the function of XM3A2 is the same as the function of M3A2. The XM3A2 device has three levels of structural hierarchy, and our test successfully found the faulty part at the lowest level. Though the example of XM3A2 is somewhat simplistic, it shows the capability of our system to deal with a wide range of devices in the domain with arbitrary complexity.

FUTURE WORK

In our current scheme, similar component types, which have the same function but different specifications, are represented individually. An example is the representation of 1-to-1, 1-to-2, and 1-to-3 transformers. It would be better to represent all types of transformers by a single representation with a parameter to specify the transforming ratio. There is also no specific user interface for adding new components so far.

Our future plans include the investigation of domain knowledge, and further development of the knowledge representation scheme for reasoning and display. We also have noted interesting differences between structural and functional displays which we will follow up.

ACKNOWLEDGEMENT

We would like to thank Bill Rapaport for reading an initial draft of this paper.

References

Davis85.
Randall Davis, Howard Shrobe, and al., "The Hardware Troubleshooting Group," *SIGART Newsletter* **93** pp. 17-20 (Jul. 1985).

Davis84.
R. Davis, "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence* **24** pp. 347-410 (1984).

Fikes85.
R. Fikes and T. Kehler, "The Role of Frame-Based Representation in Reasoning," *Communications of the ACM* **28(9)** pp. 904-920 (Sep., 1985).

Genesereth84.
M. R. Genesereth, "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence* **24** pp. 411-436 (1984).

Hartley84.
R. T. Hartley, "CRIB: Computer Fault-finding Through Knowledge Engineering," *Computer*, pp. 76-83 (March, 1984).

Hendrix79.
Gary G. Hendrix, "Encoding Knowledge in Partitioned Networks," pp. 51-91 in *Associative Networks: The Representation and Use of Knowledge by Computers*, ed. Nicholas V. Findler, Academic Press, New York (1979).

Latto84.
Andrew Latto, David Mumford, and Jayant Shah, *The Representation of Shape*, IEEE (1984).

Shapiro79.
S. C. Shapiro, "The SNePS Semantic Network Processing System," pp. 179-203 in *Associative Networks: The Representation and Use of Knowledge by Computers*, ed. Nicholas V. Findler, Academic Press, New York (1979).

Shortliffe76.
E.H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*, American Elsevier/North Holland, New York (1976).