Reprinted from PROCEEDINGS OF THE ANNUAL RELIABILITY AND MAINTAINABILITY SYMPOSIUM, Philadelphia, PA, January 27-29, 1987

Knowledge Based Modeling of Circuit Boards†

Mingruey R. Taie; SUNY at Buffalo; Buffalo

James Geller; SUNY at Buffalo; Buffalo

Sargur N. Srihari; SUNY at Buffalo; Buffalo

Stuart C. Shapiro; SUNY at Buffalo; Buffalo

Key Words: Maintenance, Fault Diagnosis, Expert System, Device Modeling, Device Representation, Graphical Knowledge, Circuit Board Analysis, User Interface, Knowledge Based Graphics, Knowledge Representation.

Abstract

This paper describes a maintenance expert system that has been designed with a focus on applied knowledge representation. Two main points of interest are described, the representation and reasoning mechanisms necessary for diagnosis based on a deep model of a device, and the representation for an integrated graphical user interface with limited natural language capabilities. Device structure is represented in a hierarchy of device types. Structural templates and instantiation rules permit focused diagnostic reasoning using lazy instantiation. Functional description is procedurally attached to the declarative network representation. Similarly, pieces of graphics code are attached to a declarative representation of the graphical appearance of the device.

Introduction

The VMES research project is aimed at the development of a versatile maintenance expert system for digital circuit troubleshooting. Theoretical and practical aspects of fault diagnosis, knowledge representation for system versatility, and knowledge-based graphical representation of target devices are being investigated.

VMES is designed to be versatile across a range of target devices in the chosen domain (a class of digital circuits); across most of the possible faults; across different maintenance levels; and across a variety of user interfaces. To achieve these versatilities, the device-model-based approach is followed. The device-model-based approach, as opposed to the empirical-rule-based approach used by MYCIN [Shortliffe76a] for medical diagnosis and by CRIB [Hartley84a] for computer hardware fault diagnosis, is suggested to have advantages in knowledge acquisition, diagnosis capability, and system generalization [Davis83a, Davis84a, Geneseretha].

VMES is implemented in SNePS [Shapiro79a], the Semantic Network Processing System, and has several modules: an expandable component library as its knowledge base; an inference package (part of SNePS) with diagnostic rules; an active database for diagnosis; a user interface for intermediate users (engineers) to adapt VMES to new devices by incrementally updating the component library; and a multi-media user interface for end users (technicians) to interact with VMES for fault diagnosis. The architecture of VMES is shown in Figure 1.

Since a device-model-based fault diagnosis system reasons directly on the structure and function of a device and usually uses a simple inference engine, the representation of the device is vital to system performance. We use a hierarchical representation of knowledge to provide abstraction levels of devices. This allows a fault diagnosis system to focus on either individual objects or on several objects at a time.

The knowledge base stores descriptions of all component types used by the target devices. Objects which are parts of a device are instantiated only when needed. A formalism for device representation using instantiation rules and structural templates has been designed to describe the structure, the function, the intended maintenance level, and the test instruction of each component type to the knowledge base. The basis of the inference engine is SNIP, the SNePS Inference Package. It also includes an algorithm and diagnostic rules for carrying out the diagnosis. The active database is created for each diagnosis to store instantiated objects and their associated port values and states.

The multi-media user interface has menu, graphical, and limited natural language capabilities. Most human dialogues about a technical subject profit from the use of diagrams in addition to natural language communication. For circuits, technicians typically use wire plans which represent the function of a board, and structural diagrams which show the physical layout of the components of the circuit. The most natural way for a maintenance system to ask a user about the voltage value at a specific location is to display the structural diagram of the board and mark the position which is currently under focus, for instance by highlighting it. While CAD systems with similar interfaces exist, they are typically based on data objects like points, lines, etc. and do not store knowledge about visual properties of the *domain objects* that they are dealing with. In our approach all information necessary to display an object is stored as *knowledge* in a common framework with the knowledge necessary for doing diagnosis.

In this way we have combined our work on the graphical interface with more basic research in the representation of graphical knowledge. Systems of knowledge-based graphics have been reported in the literature by Zydbel et al [Zydbel81a]. and by Friedell [Friedell84a] By incorporating more and more knowledge into the graphical interface we have attained a new level of graphics that we want to refer to as *Intelligent Machine Drafting*.

Device Modeling

Compact representation is desirable for memory economy, diagnostic efficiency, and system versatility. In observing that many parts of an electronic device may have the same component type and thus show the same function, we find that representing every detail of a device will create unnecessary redundancy, which impairs system performance and versatility. Instead of representing all objects explicitly, VMES maintains an expandable component library, and objects are instantiated as needed. Devices are modeled hierarchically, and objects, which may be the device itself or its sub-parts at any hierarchical level, are represented as modules. Several implementations have been experimented with [Shapiro86a, Taie86a], and a formalism which represents devices by instantiation rules and structural templates is described here.

Structural Knowledge

The component library consists of descriptions of all component *types* used to construct the devices at all hierarchical levels. Each component *type* is in turn abstracted at two levels, and represented by a SNePS rule and a SNePS assertion. The former is categorized as an "instantiation rule", and the latter a "structural template".

At level-1 abstraction, knowledge about a component type is represented as an instantiation rule. The rule is used to instantiate an object of the component type as a module with I/O ports and associated functional description. The functional description is implemented as a LISP function that calculates the desired port value in terms of the values of other ports; this allows the simulation of the device.

At level-2 abstraction, a structural template is used to describe the sub-parts and wire connections of the object at the next hierarchical level. Component types and intended maintenance levels of sub-parts are also indicated. A structural template provides the necessary knowledge about the sub-structure of all objects of the same component *type* without representation overhead. Unlike instantiation rules, structural templates are never executed (fired) to produce a representation

† This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB DC 20332 under Contract No. F30602-85-C-0008.

for any specific object. When reasoning on the sub-structure of an object is required, instead of instantiating the sub-structure (all the sub-parts and wire connections) and then reasoning on the resulting representation, we do it directly on the structural template of the object, and only suspicious sub-parts are instantiated for further examination.

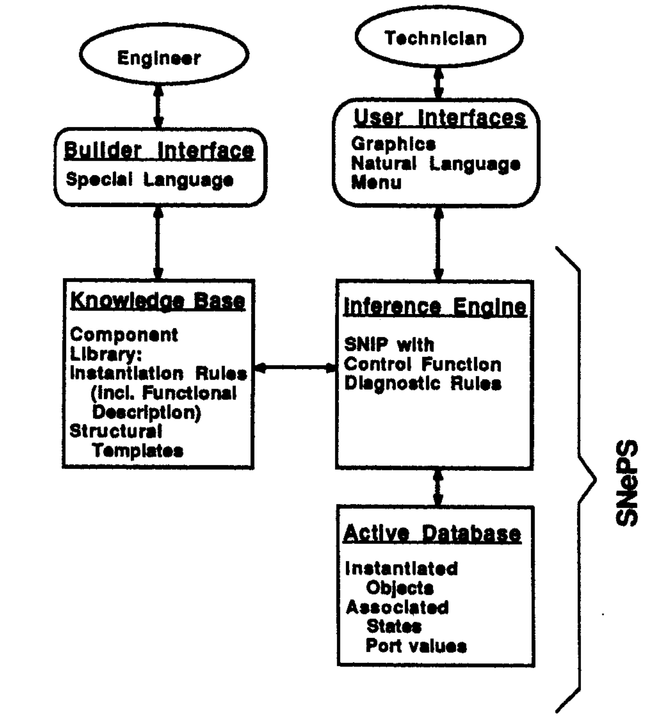

Figure 1. Architecture of VMES

All annotations are shown in italics.

```
(build avb $x
    ant (build object *x type PCM6 abs-lv IRfL1)
        The antecedent part states that for every object x if
        it is of type PCM6 and to be instantiated at level-1 by
        this rule then do the consequents. The first part of
        consequents builds the i/o ports of the object. The
        second part of consequents assigns the functional
        description to the output ports. The listing here is
        incomplete due to limited space. Below are some input
        ports for timing control.
    cq (build inport-of *x id t-shift) = vTS
    cq (build inport-of *x id r-shift) = vRS
        Below are some input ports for voice signal.
    cq (build inport-of *x id siginl) = vSI1
    cq (build inport-of *x id sigin2) = vSI2
        Below are some output ports for voice signal.
    cq (build outport-of *x id sigoutl) = vSO1
    cq (build outport-of *x id sigout2) = vSO2
        Below are some function assignments of voice signal
        output. It states that to calculate the port value of
        sigout1 (*vSO1), the function PCM6sigout with five
        arguments (pn 5) which are sigin1 (p1 *vSI1) and
        other four timing signals (p2-p5) is to be used, and
        the calculated result may have 5% of error tolerance
        (tolrnc 5).
    cq (build object *vSO1 func PCM6sigout tolrnc 5
            pn 5 pl *vSI1 p2 *vTS p3 *vRS p4 *vT0 p5 *vR0)
    cq (build object *vSO6 func PCM6sigout tolrnc 5
            pn 5 pl *vSI6 p2 *vTS p3 *vRS p4 *vT8 p5 *vR8]
```

Figure 2. Instantiation Rule for Level-1 Abstraction of Component *type* PCM6. (For testing purpose, the digital i/o pcmin & pcmout are connected for each channel to form a loop so that input at sigin is echoed to the output sigout)

This model is illustrated in Figures 2 to 5, where Figures 2 and 3 are annotated SNePS codes which build the SNePS representation for the two level abstractions of a component type PCM6, and Figure 4 and 5 are their pictorial equivalents. PCM6 stands for the six channel pulse coded modulation boards which are used for telecommunication.

There are three sections of a structural templates: the first one identifies that the template is for a particular type such as PCM6; the second section described the subparts; and the last section envisions the wire connections.

```
(build
   Section 1: Structural Template Identification.
 type PCM6 abs-lv STfL2
   Section 2: Subparts Description (incomplete). The id
   part gives the unique id of a subpart within the template.
   The ext-name part is used to extend the name of the
   subpart when it is instantiated. The type part gives
   the type of the subpart, and the mntn-lv part indicates
   its intended maintenance level.
 sub-parts
    ((build id PCM6-pc1 ext-name PC1 type PCC
           mntn-lv DEPOT)
     (build id PCM6-pc2 ext-name PC2 type PCC
           mntn-lv DEPOT)
     (build id PCM6-nt0 ext-name NOTG.t0 type NOTG
           mntn-lv DEPOT)
     (build id PCM6-nt1 ext-name NOTG.t1 type NOTG
           mntn-lv DEPOT)
     (build id PCM6-ix1 ext-name IX1 type XFORM
           mntn-lv DEPOT)
     (build id PCM6-ix2 ext-name IX2 type XFORM
           mntn-lv DEPOT))
   Section 3: Wire Connections. (incomplete)
 connections (
      voice signal side ...
     (build from (build inport-of  PCM6    id siginl)
           to  (build inport-of  PCM6-ix1 id in))
     (build from (build outport-of PCM6-ix1 id out)
           to  (build inport-of  PCM6-pc1 id sigin))
      pcm i/o side ...
     (build from (build inport-of  PCM6    id pcmin-A)
           to  ((build inport-of  PCM6-pc1 id pcmin)
                (build inport-of  PCM6-pc2 id pcmin)
                (build inport-of  PCM6-pc5 id pcmin)))
      timing control ...
     (build from (build inport-of  PCM6    id t0)
           to  (build inport-of  PCM6-nt0 id in))
     (build from (build outport-of PCM6-nt0 id out)
           to  ((build inport-of  PCM6-pc1 id t-strobe)
                (build inport-of  PCM6-pc3 id t-strobe]
```

Figure 3. Structural Templates for Level-2 Abstraction for Component *type* PCM6.

Functional Knowledge

Functional knowledge of a component *type* is represented as a procedural attachment to the semantic network. The functional description is usable to simulate the component behavior, i.e., to calculate the

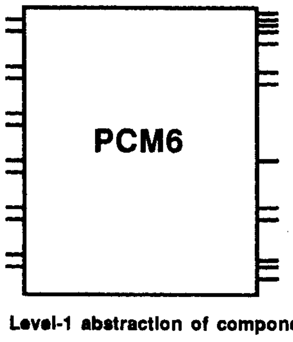

Figure 4. Level-1 abstraction of component type PCM6.

values of output ports if the values of the input ports are given. It should also be usable 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 instantiation rule of the component *type*. Since different ports of different component types might display the same behavior, some functions can be shared. Figure 6 shows some examples of functional description.

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

```
(defun ADDERout (inp1 inp2)
        (+ inp1 inp2))
```

Below is an example to show a function shared by several different component types namely by the type "PCM6", 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)
```

Below is the function for the output port of AND.Gate-type objects. (Other coding is possible, but since we use 1/0 for high/low, the following way is a convenient one.)

```
(defun ANDGout (inp1 inp2)
        (/ (+ inp1 inp2) 2))
```

Figure 6. Examples of Functional Description.

Graphical Knowledge

While the process of diagnosis is running, the user is informed about the activities of the system via a graphical trace of its reasoning.

The device is displayed on a graphics terminal, and parts currently under consideration are highlighted, for instance, by changing their color. Some more details about this will follow in a later section, here only representational questions will be raised.

In the chapters above we have introduced a notation for the knowledge used for diagnosis, which is based on the SNePS user language. All the information necessary to create a graphical representation of the diagnostic object is stored in the very same knowledge representation environment. This not only means that we are using the same SNePSUL syntax to describe objects in a way that pictures can be created, but we are using a common knowledge base, and in fact to a certain degree the *same knowledge* for the diagnosis and the drawing programs.

In this representation only primitive shapes are stored in a form comparable to "classical" graphics programs. For instance, the form of a multiplier is stored as a piece of code that, if executed, draws a multiplier. However any more complicated entity is stored declaratively in the network representations.

It is necessary to distinguish between two different types of graphical representations which pose different requirements. In a structural or physical representation a device is shown in a geometrically analog way. If a resistor is below a chip on a board, then one can expect the picture of a resistor below the picture of a chip on the corresponding plan. In order to construct such a plan a human as well as a system must have positional knowledge. This knowledge is usually expressed by coordinate pairs.

Functional or logical representations, on the other hand, do not need positional knowledge. If one draws a wire plan, it is not necessary to know exactly where to put a component. Certain connectivity conditions have to be satisfied in order to create a picture true to the object, and certain conventions of the draftsman's trade have to be observed, but there are no a priori rules that specify that a certain resistor must be under a certain chip. In fact not even neighborhood relations have to be preserved.

We will first describe the representation used for structural descriptions, and then talk about logical descriptions (wire plans). All the routines for structural display have been implemented, and the implementation for functional display is currently well on its way.

Graphics Structural Descriptions

It is necessary to know about the *form* of every object involved in the production of a drawing. In our system, forms are either linked directly to the corresponding object or an object inherits a form from a class of objects. This requires two case frames, one linking the object to a class and a second one linking the class to a form. Forms represent the link between the declarative and the procedural plane of the representation system. A form is at the same time two different things: it is a (base) node in the semantic network, and in this way accessible by the knowledge base handler, but it is also the name of a LISP function that contains calls to routines of a LISP graphics package. SNePSUL expressions for some of the caseframes described in this and the next section are given in Figure 7.

Positions are represented in a variety of different ways, the main ones being absolute positions, relative positions and relative sub-part positions. The caseframe for an object which is located at an absolute position contains an arc (a slot) to the object, and arcs to nodes denoting coordinate numbers. In the case of a relative position an additional arc identifies a "reference object". In order to draw a relatively placed object, the drawing program has to retrieve the position of the reference object first.

Relatively placed sub-parts do not have an arc to a reference object. In this case, the assumption made by the drawing system is that this part must be placed relative to its "super-object". In general the position of any part is assumed to be the position of its reference point. By convention the upper left corner is made the reference point whenever possible. So the caseframes described above relate to the reference points of the involved objects.

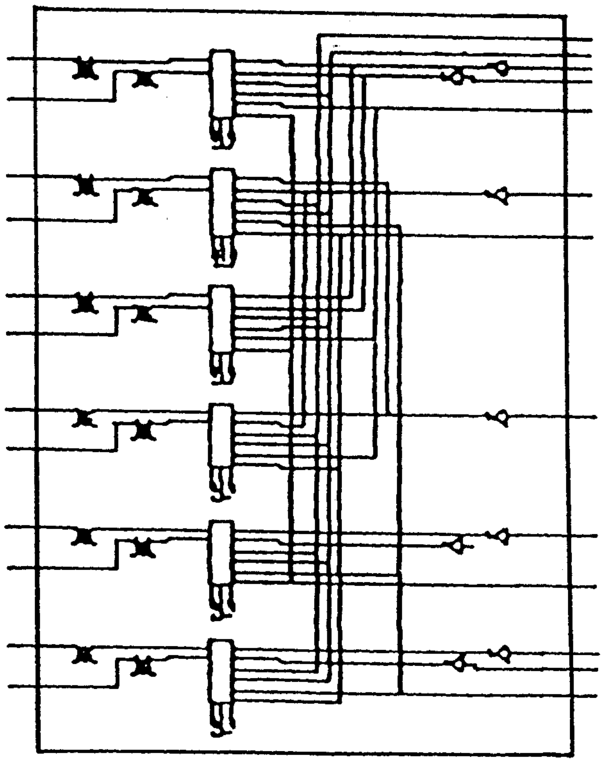

Figure 5. Level-2 abstraction of component type PCM6.

```
(build object D1-M1
        form  xmult
        modality function)
```
This describes an object with Individual Form

```
(build object D1-M2
        type  multiplier
        modality function)
```
This asserts that D1-M2 is a multiplier

```
(build class multiplier
        form  xmult
        modality function)
```
This expression links the class multiplier to the form xmult

```
(build object D1
        sub-parts (D1-M1 D1-M2)
        sub-assems (build inport-of D1
                          id       inp1)
        abspos (build x 100 y 200)
        modality function)
```
This is a partial description of D1. It has 2 parts D1-M1 and D1-M2, one sub-assembly which is an input port with the id inp1 and an absolute position at 100/200

Figure 7. Example Caseframes for Graphical Knowledge.

Our system also permits us to assign *attributes* to objects. We are discriminating between iconic and symbolic attributes. Iconic attributes are directly displayable, for instance "color" is such an attribute. Symbolic attributes require a mapping function that assigns a displayable attribute to a symbolic attribute. The diagnosis program makes heavy use of this possibility. For instance, the state (symbolic) attribute is displayed by mapping it to the color (iconic) attribute.

We are viewing attributes as functionals that take a form *function* and an attribute-value as arguments and return a modified form function. The mapping between symbolic attribute values and iconic attribute values is therefore done procedurally. For the example above the state "faulty" would be mapped to the color "red", and the state "suspect" would be mapped to "green".

Besides the class hierarchy, a *part hierarchy* is also employed using a caseframe with an object arc and one or more sub-parts arcs. It is possible for the user to specify how many levels of the part hierarchy he wants to see displayed.

Graphics Functional Representations

If one tries to create logical (functional) representations from a knowledge base, then the following interesting points become notable:

-- All the positional information can be eliminated. It should not be necessary to specify any location.

-- In order to create a reasonable picture, more knowledge of other types is necessary. For example, it is necessary to know which objects are inports and which objects are outports, because people usually expect the signal flow on a diagram to go from left to right, or from top to bottom. Incidentally, this information is also necessary for diagnosis, and therefore does not create any additional requirements.

-- Forms loose their absolute meaning. Objects like wires and boards especially don't need to be specified by graphics code. In fact the forms of all wires will be the result of a routing algorithm. It is a well known discriminating factor between declarative and procedural representations that the latter do not permit incomplete specifications, while the former ones do. But this is exactly what one would like to have. All wires consist of vertical and horizontal lines, but nothing about the specific form of a wire is known before the positions of the impinging components have been fixed.

We also have found that, in general, for either type of representation:

-- the classical part hierarchies are not sufficient even for most primitive applications. We have so far discriminated between real parts and sub-assemblies, and one more type of whole-part hierarchy for clusters might become necessary. Sub-assemblies are represented very similarly to sub-parts, only the arc sub-parts is replaced by the arc sub-assems.

The AI literature has so far discussed inheritance along the lines of a class hierarchy. We have found it necessary to do inheritance along our part hierarchy, and to control this inheritance with a meta-attribute. In this way a user can state declaratively whether he wants an attribute inherited or not. While we *do* want to inherit that a big chip has big ports we do *not* want to inherit that if a board is faulty, then so are all its parts!

Using no spatial information about the location of objects forces us to deal with placing and routing algorithms, however our objectives are far from a VLSI designer, and more comparable to the TYGES project [Eades86a] in that we are trying to create a graphically pleasing representation. We refer to this type of drawing activity as *Intelligent Machine Drafting*, a term that we have not yet seen in the literature.

We have defined a very limited class of circuits and are working on automatic placing of members of this class. The class has been defined formally, but in this paper we will limit ourselves to an intuitive summary description. Devices in the class consist of 0 or 1 main object and of parts of this main objects. Every part, as well as the main object, has "ports", and ports are connected by wires. The number of signal paths and the length of signal paths is expected to be small enough to permit placing them with our *column-equal-spacing* algorithm (which will be reviewed later). All components have to be connected "straight forwardly", i.e. no feedback loops are permitted. Icons (= parts) are all about the same size.

The column-equal-placing algorithm classifies elements according to their signal distance from the system input ports and assigns every part to a column. It then equally distributes all columns over the screen, and equally distributes all parts inside their respective columns. We are currently reviewing the literature on VLSI routing, in order to decide upon an appropriate routing algorithm.

The advantage of a system that does its own placing based on information about the structure of the device is obvious -- to a large extent, creating a knowledge base for maintenance purposes takes care of creation of the necessary graphics. We have also found knowledge based programming to be far more robust than all other common programming paradigms. The reasoning system and the graphical interface of VMES were designed by different people with a minimum amount of personal interaction, nevertheless system integration did not pose any difficulties.

Another significant advantage of our representation system for graphical knowledge is the large amount of SNePS natural language processing software that can immediately be used. We have used the SNePS ATN package [Shapiro82a] to create a natural language interface to the graphics routines. This permits us to do *Natural Language Graphics* [Brown81a, Hussman84a] almost as a byproduct, the only step necessary was the creation of an ATN grammar for the graphics/circuitboard domain.

The user can for instance request from the system:

show me all multipliers
show me D1M1
show me all faulty adders
please display D1

and many others, where D1 is the name of a device, and D1M1 the name of its first multiplier.

Diagnostic Reasoning

The diagnostic reasoning of VMES 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. Though the target domain of VMES is digital circuit boards, we observed that in real life most of the electronic boards contain some simple analog components such as resistors and transformers. Therefore, for practical consideration, some components are allowed to have a tolerance when their outputs are being checked for violated expectations. The tolerance information is associated with the instantiation rule as depicted before.

After violated expectations are detected, the system uses the structural template to find a subset of components at the next lower hierarchical level which might be responsible for the bad outputs. Then instantiation rules are activated to instantiate the suspects, and the suspects are ordered by some criteria for further investigation. Suspect ordering criteria will be discussed later. The diagnostic process is then continued on the instantiated suspicious parts. A part is declared faulty if it shows some violated expectation at its output port and it is at its intended maintenance level as described in the structural template or it is at the bottom level of the structural hierarchy and no further diagnosis is possible.

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. A measured port value can either be deduced from wire connections or requested from the user. 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 in Figure. 8 in both SNePS code and in English.

Suspects are first sorted into sublists by **global criteria** called **fault possibilities**. Fault possibility is determined by evaluating the suspects against the wholistic current situation, which is the current test results. For the current implementation, there is only one global criterion: a suspect has higher fault possibility if it contributes to more vio-expct output ports. Suspects within each sublist are then sorted by some **local criteria** called **fault potentialities**. Fault potentiality is a measure of the rate a particular type of component may fail. It is independent of the environment, only depending on the component type. (It may also depend on the lot number of the component, but so far we do not treat such details.) The ideal fault potentiality data for our domain is the thermal analysis data of the components. Due to the unavailability of the thermal analysis data, it is now implemented as an index ranging form 1 to 3. Component types with no stored fault potentiality data default to 2.

VMES does not make the single fault assumption. The system incorporates the user's judgement by offering him an opportunity to terminate the diagnosis session whenever a faulty part is located. The user can choose to continue the investigation of the remaining suspects if he feels that more faults are possible or if he would merely like to make sure other suspects have no problems.

Graphical Infertrace

As mentioned in an earlier section, there is a part of the VMES system that permits the user to graphically trace the whole reasoning process. This is done by a function called *display*, that retrieves knowledge about how objects look and how they are located from the network, and computes and creates a graphical representation from this knowledge. This method is analogous to the generation of natural language from a knowledge base, a widely accepted AI technique.

The reasoning part calls display with the name of the object that should be displayed, possibly with one or more of a number of options. We will give a quick review of the possible options; more details can be found in [Shapiro86a].

It is possible to select how may levels in the part hierarchy should be displayed. Objects can be shown blinking, and they can be blown up or shrunk to fill the screen or a predefined window optimally. It is possible to create two pictures, a detailed picture of an object and a picture of the "environment" of that object. The user has only to specify the object itself; the environment is retrieved from the part hierarchy by searching upwards.

A sub-option of the environment option permits the user to limit how may levels up in the part hierarchy is searched. The selection of what to show can be limited not only by the number of levels, but also by the number of parts or according to an approximate cognitive complexity which we are simulating by counting the number of graphics primitives visible on the screen.

Conclusions

The representation scheme described in this paper has been used to represent several devices, including several multiplier/adder boards and a six-channel PCM (Pulse Code Modulation) board for telephone communication. VMES has been successful in isolating the faults on these boards. A typical example is that VMES identifies an inverter on a PCM6 board as a faulty part, which actually accounts for the simultaneous malfunctioning of the two channels it affects, in the early stage of diagnosis. Though VMES has no capability to conclude that it is the only fault on the board, the suspect ordering criteria help the system to decide which suspect is to be checked first. The result shows that the representation scheme, along with an expandable component library leads to several important advantages: compact representation and system efficiencies in both system development and operating phases.

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

Figure 8. A diagnostic rule.

We first claim that a clear distinction between the two abstraction levels of an object is desirable. The separation leads to system efficiency since the knowledge at the two abstraction levels are used at different stages of diagnosis. Level-1 information is used for detecting violated expectations, and level-2 information is used for suspect generation. To mix these two levels together will cause representation overhead and hamper system performance.

The use of the passive structural templates, which are never executed, to represent the substructure of objects of a component *type* has advantages over a procedural representation which uses a procedure or an instantiation rule for it [Davis83a, Shapiro86a]. Whenever it is necessary to reason about the substructure of an object, it is carried out on the unique structural template for the component *type* of the object. Only the sub-parts that require further examination will be instantiated (by the proper instantiation rules for them). Unlike the structural template representation, a procedural representation is used to instantiate "all" sub-parts of an object, and then the reasoning is carried out over the resulting substructures. This leads to serious system inefficiency due to representation explosion and resource waste caused by unnecessary object instantiation.

The most important feature of VMES is its versatility. VMES can easily be adapted to new devices by merely adding the structural and functional information of the "new" component *types* to the component library. A new component *type* is defined as a component *type* which has not previously 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 a device is designed.

We have found at presentations that the graphics interface considerably improves the understandability of the reasoning process of the system. The use of a knowledge based graphics system promises to simplify the creation of graphics for new devices, in this way aiding the versatility of the system. The common representation for diagnosis, graphics and a number of natural language tools has aided us in adding a natural language component to the system, and in this way strengthened our belief in the usefulness of a knowledge based graphics system as a natural interface for a user friendly maintenance expert system.

As a spin off, we have found limitations in the classical part hierarchy and inheritance mechanisms, and we have started to work on a module for *Intelligent Machine Drafting* as a part of our system.

References

Brown81a.
D. C. Brown and B. Chandrasekaran, "Design Consideration for Picture Production in a Natural Language Graphics System," *Computer Graphics* **15**(2) pp. 174-207 (July 1981).

Davis83a.
R. Davis and H. Shrobe, "Representing Structure and Behavior of Digital Hardware," *Computer*, pp. 75-82 (Oct. 1983).

Davis84a.
R. Davis, "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence* **24** pp. 347-410 (1984).

Eades86a.
P. Eades and A. J. Lee, "Perception of Symmetry," 67, University of Queensland, St. Lucia (March 1986).

Friedell84a.
M. Friedell, "Automatic Synthesis of Graphical Object Descriptions," *Computer Graphics* **18/ N.3**(1984).

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

Hartley84a.
R. T. Hartley, "CRIB: Computer Fault-finding Through Knowledge Engineering," *Computer*, pp. 76-83 (March 1984).

Hussman84a.
M. Hussman and P. Schefe, "The Design of SWYSS, a Dialgue System for Scene Analysis," in *Natural Language Communication with Pictorial Information Systems*, ed. Leonard Bolc, (1984).

Shapiro79a.
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).

Shapiro82a.
S. C. Shapiro, "Generalized augmented transition network grammars for generation from semantic networks," *The American Journal of Computational Linguistics* **8**(1) pp. 12-25 (1982).

Shapiro86a.
S. C. Shapiro, S. N. Srihari, M. R. Taie, and J. Geller, "VMES: A Network-Based Versatile Maintenance Expert System," pp. 925-936 in *Proc. of 1st International Conference on Applications of AI to Engineering Problems*, , Southampton, U.K. (April 1986).

Shortliffe76a.
E. H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*, American Elsevier/North Holland, New York (1976).

Taie86a.
M. R. Taie, S. N. Srihari, J. Geller, and S. C. Shapiro, "Device Representation Using Instantiation Rules and Structural Templates," pp. 124-128 in *Proc. of Canadian AI Conference - 86*, , Montreal, Canada (May 21-23, 1986).

Zydbel81a.
F. Zydbel, N. R. Greenfeld, M. D. Yonke, and J. Gibbons, "An Information Representation System," *IJCAI - 81*, (1981).

Biographies

Stuart C. Shapiro
Department of Computer Science
226 Bell Hall, Amherst Campus
State University of New York at Buffalo
Buffalo, NY 14260, USA

Stuart C. Shapiro received the S.B. degree in mathematics from MIT in 1966, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin, Madison in 1968 and 1971, respectively.

He currently holds the positions of Professor and Chairman of the Department of Computer Science at the University at Buffalo, where he has been since 1977. In 1971, he was a Lecturer in Computer Sciences at the University of Wisconsin, Madison. Between then and going to Buffalo, he was at the Computer Science Department of Indiana University, as Assistant and Associate Professor. In summer, 1974, he was a Visiting Research Assistant Professor at the University of Illinois at Urbana-Champaign.

Dr. Shapiro's research interests are in artificial intelligence, natural language processing, knowledge representation, and reasoning. He is editor of *The Encyclopedia of Artificial Intelligence* (John Wiley & Sons, forthcoming), the author of *Techniques of Artificial Intelligence* (D. Van Nostrand, 1979), *LISP: An Interactive Approach* (Computer Science Press, 1986), and over 60 technical articles and reports. He has served as a consultant on Artificial Intelligence for several companies, as department editor of the journal, *Cognition and Brain Theory* for Artificial Intelligence, and has served on the editorial board of the *American Journal of Computational Linguistics*.

Dr. Shapiro is a member of the ACM, the IEEE, the ACL, the Cognitive Science Society, and the AAAI. He is listed in *American Men and Women of Science*, and in *Who's Who in Artificial Intelligence*.

Sargur N. Srihari
Department of Computer Science
226 Bell Hall, Amherst Campus
State University of New York at Buffalo
Buffalo, NY 14260, USA

Sargur N. Srihari received his Bachelor's degree in Electrical Communication Engineering from the Indian Institute of Science in 1970 and Ph.D. degree in Computer and Information Science from the Ohio State University in 1976. He is presently a tenured associate professor in the Computer Science department at SUNY Buffalo. In summer 1979, he was a visiting Research Assistant Professor at the University of Waterloo in Canada. His current research interests are in artificial intelligence, and in particular, knowledge-based systems for diagnosis and computer vision.

Dr. Srihari has published over 60 journal articles and conference papers. He is the author of *Computer Text Recognition and Error Correction* (IEEE Computer Society Press, 1984). He is presently an associate editor of the Pattern Recognition journal. He is on the advisory board of the Second AI Applications to Engineering Problems Conference to be held at MIT in 1987.

Dr. Srihari is a senior member of IEEE, and member of the Association for Computing Machinery, American Association for Artificial Intelligence and Pattern Recognition Society.

James Geller
Department of Computer Science
226 Bell Hall, Amherst Campus
State University of New York at Buffalo
Buffalo, NY 14260, USA

James Geller is a doctoral candidate in the department of Computer Science at the State University of New York at Buffalo. He received a "Dipl. Ing." in Electrical Engineering from the Technical University Vienna, Austria in 1979 and the MS in Computer Science from the State University of New York at Buffalo in 1984. His current research interests are in artificial intelligence, and in particular, knowledge representation for pictorial domains.

Mingruey R. Taie
Department of Computer Science
226 Bell Hall, Amherst Campus
State University of New York at Buffalo
Buffalo, NY 14260, USA

Mingruey R. Taie is a doctoral candidate in the department of Computer Science at the State University of New York at Buffalo. He received a BS in Mechanical Engineering from the National Taiwan University in 1978 and the MS in Computer Science from the State University of New York at Buffalo in 1984. His current research interests are in artificial intelligence, and in particular, knowledge representation and fault diagnosis.