VMES: A NETWORK-BASED VERSATILE MAINTENANCE EXPERT SYSTEM

Annual Report on Contract No. F30602-85-C-0008 for 1985

submitted to

Northeast Artificial Intelligence Consortium
120 Hinds Hall
Syracuse University
Syracus, NY 13210

November 25, 1985

submitted by:

Stuart C. Shapiro, Principal Investigator
Sargur N. Srihari, Co-Principal Investigator
James Geller, Graduate Research Assistant
Ming-Ruey Taie, Graduate Research Assistant
Scott S. Campbell, Graduate Research Assistant

Department of Computer Science
State University of New York at Buffalo
226 Bell Hall
Buffalo, NY 14260

Part. I Device Modeling and Fault Diagnosis of VMES

1. INTRODUCTION

We are developing a versatile maintenance expert system (VMES) for trouble-shooting digital circuits.

Some diagnosis systems, such as MYCIN [19] for medical diagnosis and CRIB [5] for computer fault diagnosis, are built on rules which represented 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 they have almost no capability of system generalization.

As a solution to difficulties of empirical-rule-based diagnosis systems, structural and functional descriptions have been widely used by AI researchers in the domain of fault diagnosis [3,4]. 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.

To test this idea, we have implemented a diagnosis system that has successfully pinpointed the faulty part of a multiplier/adder board, a favorite example for researchers in this field (see e.g., [3].)

An important aspect of our research is to find a good knowledge-representation scheme to support the diagnosis and the construction of a versatile maintenance system. We have implemented our system in the SNePS Semantic Network Processing System [16]. Advantages are: (1) structural and functional knowledge is integrated into a single network; (2) reasoning is done by rule-based inference; (3) diagnosis assumptions can be handled in a natural way; (4) the deduction process can be monitored; (5) inference can also be traced graphically; (6) the representation can easily be expanded and modified; (7) procedural knowledge is represented and used; (8) it is smoothly interfaced with LISP.

Version 1 of our implementation uses a hand-coded description of the device. An intermediate user, who adapts the system to a specific device, needs to hand-code all the structural and functional details of the device, even that a lot of parts are of the same component type. Since versatility is a goal, the system was redesigned as Version 2. It contains a kind of type declaration to build a component library. This has enhanced the versatility of the system quite significantly. We successfully adapted the system to a new device with minimum effort by just adding the descriptions of new components to the system.

A brief description of the implementation of Version 1 appears in the next section. Section 3 contains a detailed description of our current implementation (Version 2) along with an annotated demonstration. Section 4 and 5 are discussions and future work.

2. VERSION 1

This section contains a brief description of our early implementation of VMES. A board of three multipliers and two adders was used as the target object to be diagnosed.

The structural description was hierarchical, which made it possible to focus on the relevant part of the device at any time during the diagnosis. The structural description was hard-wired, every detail of the device needed to be entered by hand. Examples are:

```
   In SNePS codes                                    In English

(build  object      D1                         The object D1 is of M3A2 type;
        type        M3A2                       it has three inputs and two outputs
        inp1        D1inp1                     named in order as D1inp1, D1inp2,
        inp2        D1inp2                     D1inp3, D1out1 and D1out2;
        inp3        D1inp3                     it is consists of 5 sub-parts: D1M1,
        out1        D1out1                     D1M2, D1M3, D1A1 and D1A2.
        out2        D1out2
        sub-part    (D1M1 D1M2 D1M3
                     D1A1 D1A2)
        super-part  NIL)

(build  object      D1M1                       The object D1M1 is a MULTiplier,
        type        MULT                       it has two inputs and one output
        inp1        D1M1inp1                   named in order as D1inp1, D1inp2,
        inp2        D1M1inp2                   and D1out1. It has no sub-part and
        out1        D1M1out1                   it is part of D1.
        sub-part    NIL
        super-part  D1)

(build  from        D1inp1                     There is a wire connection from
        to          (D1M1inp1 D1M2inp1))       D1inp1 to D1M1inp1 & D1M2inp1.
```

Only one wire connection description, which actually represented two wires, is shown here. And similar codes for D1M2, D1M3, D1A1 and D1A2 were required since the structural description was hand-coded.

Functional definition was implemented as a template of the SNePSUL function node [17]. Unlike the structural description, the functional description was associated with each type of the components rather than the parts themselves. An example was:

In SNePS code:

```
(dp ADDER (inp1 inp2 out1)
    (cond ((eq (plus inp1 inp2) out1) (succeed true))
          (t (succeed false))))
```

In English:

For an adder, it is good if the two inputs sum to the output,
it is bad otherwise.

The function description gave an explicit definition to decide whether a component was malfunctioning. It did not explicitly depict what the function of an adder was, which was required in order to simulate the behavior of an adder. And for every type of component, it needed its own rule for finding violated expectation at the output ports. The rules, in English, looked like:

If an object is an adder and all its input and output values are known, then one and only one of the following is true:
1). the object is functioning well, which can be infered from the adder function description;
2). there is a violated expectation at its output.

The inference engine for fault diagnosis followed a simple control structure. It is similar to that of the current implementation, and is discussed in the next section.

3. VERSION 2

This section contains a full description of our current system implementation. The system consists of two major parts: the device representation and the inference engine. The device representation is further divided into structural and functional descriptions. An annotated demonstration of the system is at the end of this section.

3.1. Device Representation

The current implementation of VMES includes a complete redesign of the device representation in both the structural and functional description. The disadvantages of the hand-coded description have been removed, and major progress has been made toward an ultimately versatile system.

3.1.1. Structural Description

Once again, only the logical structure of the device is represented and used for diagnosis in our current implementation. Instead of hand-coding every detail of the device, the system keeps a component library which know 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 structure of the device is still represented in a hierarchical way through the parts hierarchy. Sub-parts of the 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 is built, and the wire connections between the object and its sub-parts, as well as those among the sub-parts themselves are made. 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.

Several typical instantiation rules are as follows:

All annotations are shown in italics.

All types of components are described as:
level-1 description: I/O ports and functions.
level-2 description: sub-parts and connections.

```
; The following two SNePS rules describe
; the M3A2 type components:

(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 says that "if x is an M3A2 and is to be
;   instantiated at level-1 (TBI-L1), then do the 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 "to simulate the
;   value of first output, uses the function M3A2out1 which takes
;   three parameters: the inputs of the object x in order."
; Similar links can be done for all input ports if we want to infer
;   their values from other i/o ports.

(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 TBI-L2, uses the
;    function Give-PID-M3A2 to get the names for its sub-parts"
; The next seven lines declare the types of the sub-parts and will
;    activate appropriate rules to instantiate them at their level-1
;    instantiation. The super-part/sub-parts hierarchical relation
;    between the object x and its sub-parts is built also.
; The remainders connect the wires between x and its sub-parts as
;    well as those among the sub-parts themselves.



; The following two SNePS rules describes
; the MULTiplier type components:

(build
   avb $x
   ant (build object *x type MULT state TBI-L1)
   cq (build inport-of *x inp-id 1) = vINP1
   cq (build inport-of *x inp-id 2) = vINP2
   cq (build outport-of *x out-id 1) = vOUT1
   cq (build port *vOUT1 f-rule MULTout1
              pn 2 p1 *vINP1 p2 *vINP2]

(build
   avb *x
   ant (build object *x type MULT state TBI-L2)
   cq (build super-part *x sub-parts lamDRU]

; Please note that the level-2 instantiation will not instantiate any
;    sub-part since a multiplier is regarded as a Depot Replaceable
;    Unit (DRU) and there is no need to represent its details.
```

All instantiation rules are stored in a file, which is regarded as a components library. Representing the structure of a device via the instantiation rules and the use of a components library give the system several important advantages. We do not have to hand-code three almost identical multipliers on our example digital circuit board; the information is generated by the system only when required during the course of diagnosis. This should minimize the construction effort during the system development period, and should also gain some memory efficiency during diagnosis. This is especially important in a memory critical environment.

Although instantiation during diagnosis is good for memory efficiency, it is slower during diagnosis. 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 components 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 give 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 system adapter has to do is to add the structural and functional information of the "new" component types to the components library and the functions 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 should be readily available at the time when a device is designed.

3.1.2. Functional Description

In Version 1, the functional description was actually a testing procedure which could only be used to decide whether a component was malfunctioning. It had two main drawbacks: the description could not be used to simulate the behavior of the component, and every component type required its own associated SNePS rule for finding violated expectation at its output ports.

Version 1 offended a theoretical basis of fault diagnosis. It implemented the strategy:

> If the component is malfunctioning,
> there is violated expectation at its output.

But it should be the other way around:

> If some violated expectation is observed at the outputs,
> the component is malfunctioning.

And the violated expectation should be defined generically as:

> If there is a mismatch between the expected (calculated) value and the
> observed (measured) value at some output, it is a violated expectation.

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 had been 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 the functional description as well as the SNePS rule which finds the violated expectation are as follows:

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 for the single 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 such as the super-buffer, the wire or the
; 1-to-1 transformer.  All these component types show the same behavior
; at our level of component abstraction: echo the input to the output.

     (defun ECHO (inp1)
          inp1)


; The SNePS rule below is the only rule for concluding a violated
; expectation, it is actually part of the inference engine.  It is
; displayed here to show the benefit of the functional description of
; our current implementation.

     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 follows is true:
          1). vc and vm match;
          2). port p displays a violated expectation.
```

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

3.2. 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, tries to find the output ports which show a violated expectation, and then uses the structural description to find a subset of components at next hierarchical level which might be responsible for the bad outputs. The process is then mapped down to the suspicious parts, and 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, i.e. it is the smallest replaceable unit and there's no need to examine its details.

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 the fault. The LISP driving function is introduced for execution efficiency.

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 for the calculated value, and the last rule is to compare the two values to decide if there is a violated expectation. The last rule has been shown in the section on functional description.

The diagnosis strategy along with the combination of a LISP driving function and SNePS rules turns out to be very effective. The diagnosis can be monitored by the SNePS text or graphic inference trace. The graphical trace is only available for Version 1, but will be implemented for Version 2. Another new feature of Version 2 is that it warns the user if the diagnosis is incomplete due to insufficient information.

3.3. Demonstration Example

An annotated demonstration is shown below. The target device is an M3A2 type board. The board has three input ports and two output ports, and it has five sub-parts: three multipliers and two adders. The multipliers and adders are DRU's, thus the device has only two levels in its structural hierarchy. The structure of the test device D1 is shown in Fingure. 1.

```
; All annotations are shown in italics.

; Many output listings were removed and the SNePS inference trace
; was turned off so that the demonstration did not get too long.



; Run the SNePS system which is written in Franz LISP.
; The computer used is a VAX 11/750 at Dept. of CS, SUNY at Buffalo.

% sneps
Franz Lisp, Opus 38.79
Thu Sep 12 20:37:18 1985

sneps

; The SNePS prompt is the asterisk.

; Bring in all arc definitions used by the network:
* (intext ARCS)
(done reading from ARCS)
```

A TEST PROBLEM

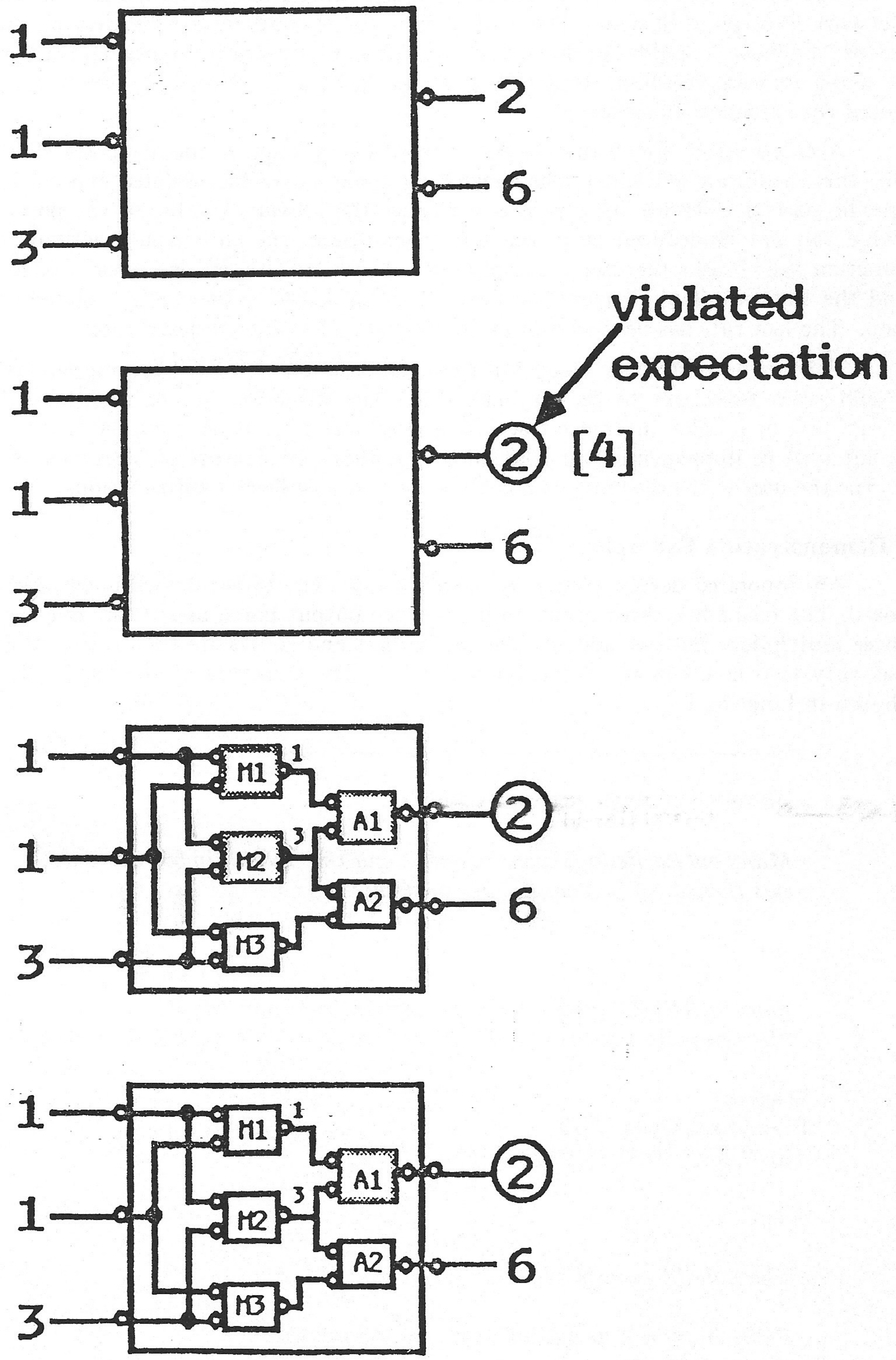

Figure 1

```
exec: 0.95 sec     gc: 0.00 sec

; Bring in the COMPonent library:
* (intext COMP)
(done reading from COMP)
exec: 19.08 sec     gc: 0.00 sec

; Bring in the CONTrol rules, i.e., the inference engine:
* (intext CONT)
(done reading from CONT)
exec: 7.03 sec     gc: 0.00 sec

; Load in the functional descriptions:
* (^(load 'FUNC))
t
exec: 1.21 sec     gc: 0.00 sec

; Declare the device D1 to be an M3A2 type object:
* (device-setup D1 M3A2)
; This activates the level-1 instantiate rule of the M3A2 type,
; and build the I/O ports and their function pointers as follows:
(m121 (state (TBI-L1)) (type (M3A2)) (object (D1)))
(m122 (inp-id (3)) (inport-of (D1)))))
(m123 (inp-id (2)) (inport-of (D1)))))
(m124 (inp-id (1)) (inport-of (D1)))))
(m126 (p3 (m122 (inp-id (3)) (inport-of (D1))))
      (p2 (m123 (inp-id (2)) (inport-of (D1))))
      (p1 (m124 (inp-id (1)) (inport-of (D1))))
      (pn (3))
      (f-rule (M3A2out2))
      (port (m125 (out-id (2)) (outport-of (D1)))))
(m128 (p3 (m122 (inp-id (3)) (inport-of (D1))))
      (p2 (m123 (inp-id (2)) (inport-of (D1))))
      (p1 (m124 (inp-id (1)) (inport-of (D1))))
      (pn (3))
      (f-rule (M3A2out1))
      (port (m127 (out-id (1)) (outport-of (D1)))))
done
exec: 32.06 sec     gc: 3.30 sec

; The follows builds the values of I/O ports of D1:
* (build port (find inport-of D1 inp-id 1)
          value 1 source measured]
* (build port (find inport-of D1 inp-id 2)
          value 1 source measured]
* (build port (find inport-of D1 inp-id 3)
          value 3 source measured]
; Next one should be 4, but a 2 is observed for this case:
* (build port (find outport-of D1 out-id 1)
          value 2 source measured]
* (build port (find outport-of D1 out-id 2)
          value 6 source measured]
```

```
; Begin the diagnosis session for device D1:
;     The messages prefixed by @@@ are from the driving function.

* (diagnose D1)

@@@ diagnose D1:  finding vio-expct .....
; If the SNePS inference trace were on, it would show that the system found
; the first output of D1 was a violated expectation, and the other was not.

@@@ adding TBI-L2 for D1 .....
; D1 is instantiated at level-2 since further investigation is needed.

@@@ adding TFS for D1 .....
; Now, a state called TFS (To Find Suspects) is added for D1.  This
; activates the rules which find suspicious sub-parts of D1:

@@@ suspects created: (D1A1 D1M1 D1M2)
; Note that D1A2 and D1M3 are not suspects.

; The diagnosis process is mapped down to each suspect:


@@@ diagnose D1A1:  finding vio-expct .....

What is the value of port
(m155 (inport-of (D1A1)) (inp-id (1)))
value/nil?  1

What is the value of port
(m156 (inport-of (D1A1)) (inp-id (2)))
value/nil?  3
; The system asks the user to supply measured values of ports if they
; can not be deduced from the wire connections:

@@@ adding TBI-L2 for D1A1 .....

@@@ D1A1 is faulty by vio-expct & DRU
; D1A1 is found to be faulty since it is a DRU and behaves abnormally.


@@@ diagnose D1M1:  finding vio-expct .....

@@@ D1M1 shows no problem


@@@ diagnose D1M2:  finding vio-expct .....

@@@ D1M2 shows no problem

; Note that both D1M1 and D1M2 are not instantiated at level-2
; since they do not show any violated expectation at their outputs.
; Also note that the values of their I/O ports are not requested since
; they can be deduced by the system.
; A final report is given by the system:
>>>>> I GOT THE FAULTY PARTS AS >>>>>
(m237 (state (faulty)) (object (D1A1)))
(dumped)
exec: 398.35 sec    gc: 74.73 sec


; The user can check all intermediate and final results:
; only a small part of it is shown below.
* (desc *nodes - *oldnodes)
(m255 (arg
        (m254 (state (vio-expct))
               (port (m150 (out-id (1)) (outport-of (D1M2))))))
      (max (0))
      (min (0)))
; The first output of D1M2 shows no violated expectation.
(m253 (source (calculated))
      (value (3))
      (port (m150 (out-id (1)) (outport-of (D1M2)))))
; The calculated value of the first output of D1M2 is 3.
(m187 (to (m125 (out-id (2)) (outport-of (D1))))
      (from (m186 (out-id (1)) (outport-of (D1A2)))))
; A wire runs from 1st output of D1A2 to 2nd output of D1.
(m166 (sub-parts (D1M1) (D1M2) (D1M3) (D1A1) (D1A2)) (super-part (D1)))
; The sub-parts of D1 are D1M1, D1M2, D1M3, D1A1 and D1A2.
(m165 (state (TBI-L1)) (type (ADDER)) (object (D1A2)))
; D1A2 is an ADDER, and has been instantiated at level-1.
(dumped)
exec: 15.25 sec    gc: 0.00 sec

* (exit)
No files updated.
%
```

4. 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. We have implemented our system in the SNePS Semantic Network Processing System [16]. Advantages are: (1) structural and functional knowledge are integrated into a single network; (2) reasoning is done by rule-based inference; (3) diagnosis assumptions are handled in a natural way; (4) the deduction process can be monitored; (5) inference can also 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 same example as the one used in the last section in both regular mode, which did the instantiation only when needed, and the pre-instantiation mode. As expected, the former was memory efficient, and the latter was good for diagnosis speed. For the example of the M3A2 type device, the latter was four times faster than the former.

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 a new device type called XM3A2 and put it into the system. The XM3A2 type has three inputs and two outputs, and only has a single sub-part which is of M3A2 type. Actually, it is a device which has an extra layer of packaging on an M3A2 type device. The M3A2 type has been known to the system, thus only the XM3A2 needed to be described to the system, and the description is two simple instantiation rules. There is no need for new functional description since the function of XM3A2 is the same as M3A2. The 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 simple, it displays the capability of the system to deal with a wide range of devices in the domain with arbitrary complexity.

5. FUTURE WORK

A potential problem is that this approach to fault diagnosis is only good for digital circuitry without feedback. There are many devices that are mixtures of digital and analog circuitries. To adapt the system to those devices may require some modification of the device representation scheme. The representation and use of second principle rules should also be introduced for better system performance.

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 rate.

There is no user interface for adding new components so far. The development of a formal language for device representation may solve this problem as well as others. The language should support all diagnosis related activities, such as device simulation and structure retrieving. It should also support the system construction and adaptation.

Part. II Graphical Interface of VMES

1. INTRODUCTION

The main mode of communication between the SNePS reasoning mechanism used for the VMES project and the user is intended to be a graphical interface. We have implemented a new version of such an interface called "SENDING". This version supersedes the version of SENDING that has been described in our final report of the Post Doctoral project (SCEEE) [18].

The first change noticeable between these two versions is the changed domain. In our effort to construct a *versatile* program we have changed the "device" of diagnosis. While the SCEEE world consisted of a small number of logic gates which were only partially connected, we have since then tackled two new devices.

The first one of these devices is a little Adder/Multiplier that has been a test object of several researchers in the field of trouble shooting. The second one is piece of a real device, a 6 channel PCM board. This change of domain however was not the crucial step.

Only insignificant improvements have been made to the representation of visual knowledge. The basic case frames are still the same as described in the SCEEE report, and so are the used relations. The significant changes in the second generation of SENDING are an improvement in speed by approximately a factor of seven of the display program, and a considerable expansion of the power of both, the display and the readform function. This section of our report will first make some general comments on "visual knowledge" and will then continue by describing the new options of "display".

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 [9,2]. 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" or of "display". Some more references on this subject are given in the special section on "Other Activities", at the end of this paper, dealing with the acquisition of background knowledge.

2. THE GRAPHICAL INTERFACE

2.1. Motivation

Why should somebody want to implement a program like "SENDING" (SEmantic Network Domain Interface Graphics)? Our interst in this interface is twofold. Currently there is a growing interest in multi media communication [14]. Technical literature would be impossible without charts, diagrams and drawings. It seems that also a dialog between a technician and an advisory expert system about a technical object like a circuit board would profit very much from a graphical component.

One can even go so far to say that diagrams are the "interlingua" of the technical literature. The display of the device under repair can be used in our system 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 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 display of objects. We are interested in representations that can be used in *reasoning*, about forms as well as for display purposes.

2.2. 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 named objects, namely as LISP functions.

Although it is not our purpose to compete with any of the very fancy existing CAD systems, considerable improvements to readform have been made over the last year. Currently work is done on the third generation of readform. Only the introduction of arcs made it possible to design most of the common logic symbols (AND and OR gates) and of transformers. In earlier described versions of SENDING a separate libarary was necessary for round objects.

Improvements currently worked on are commands that make the creation of repetitive structures easier. Also earlier defined objects can be loaded into currently built up more complex objects.

The logical counter part of 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. Displayable propositions also have to say something about form/position of an object, and the display of the proposition is done by showing the described object.

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. We are comparing this approach to graphics with language generation from an internal knowledge representation. Such a language generation program takes a semantic network as its input and generates a surface utterance from it. The difference here is, that a picture is generated.

2.3. 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. (For explanations of the SNePS terminology refer to the given reference about SNePS [16]).

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. More details on the case frames used for form/position/attributes can be found in the repeatedly quoted SCEEE report.

2.4. Special display Parameters

2.4.1. 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 [6] 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.

2.4.2. 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 that limits the displayed information to relevant objects and relations).

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. Figure 1 displays our Adder/Multiplier board at :level 2. Figure 2 shows the same Adder/Multiplier at :level 3.

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.

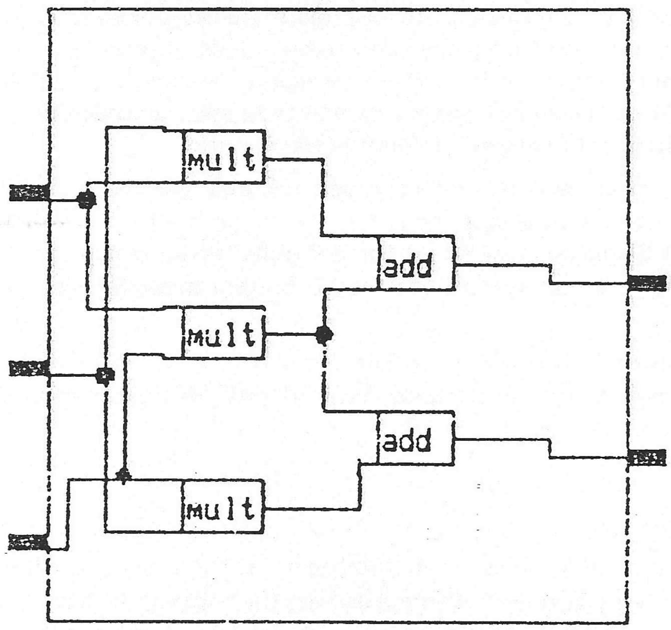

Figure 1: Adder/Multiplier at level 2

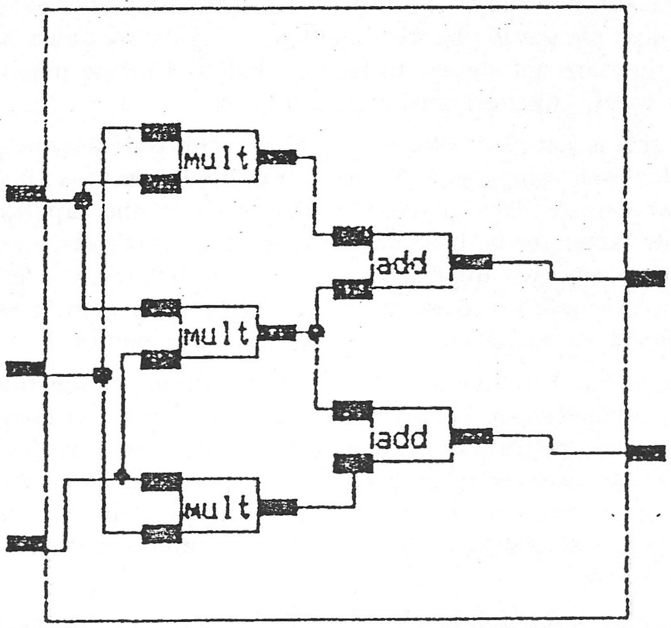

Figure 2: Adder Multiplier at level 3

Objects in the VMES system can themselves be of quite varying 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.

2.4.3. 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.

The "fill" option also permits us to avoid another common problem in computer graphics. If a window is defined arbitrarily, chances are that some of the displayed objects will be cut into two parts, only one of which is inside the window. This requires in commonly used graphics packages the very time consuming activity of "clipping". We also think that the average user is not interested in half objects. All the things that he specifies because he wants to see them, he wants to see in whole. All the objects he does not specify he either does not want to see at all, or at least he does not mind if they are not shown to him (in half!). Cutting objects into parts disagrees with our whole object oriented approach to AI.

The way all this is achieved is by having display/fill compute an optimal window in the world which complete surrounds all desired objects with the smallest possible rectangular extent. This window is mapped into the supplied viewport, using the same scale factor for both x and y coordinates. This guarantees filling the viewport in one of these two dimensions. (Note that in order to fill it in both dimensions distortions would be necessary, which might show a circle as an ellipsis. This is not only optically undesirable, but also difficult to compute).

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 pretentious, it is definitely a step towards having the system figure out what the user really wants to see as opposed as to what he is asking for. 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 into the whole".

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 ports ("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 3. Figure 3 and all other figures were created with a printer that directly dumps a screenfull from a graphics terminal. It shows a multiplier displayed in the left viewport, and the corresponding Adder/Multiplier board in the right viewport. Finally Figure 4 shows the 6 channel PCM board in the right viewport and one of its PCM chips in the left viewport.

2.5. 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 traceing 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

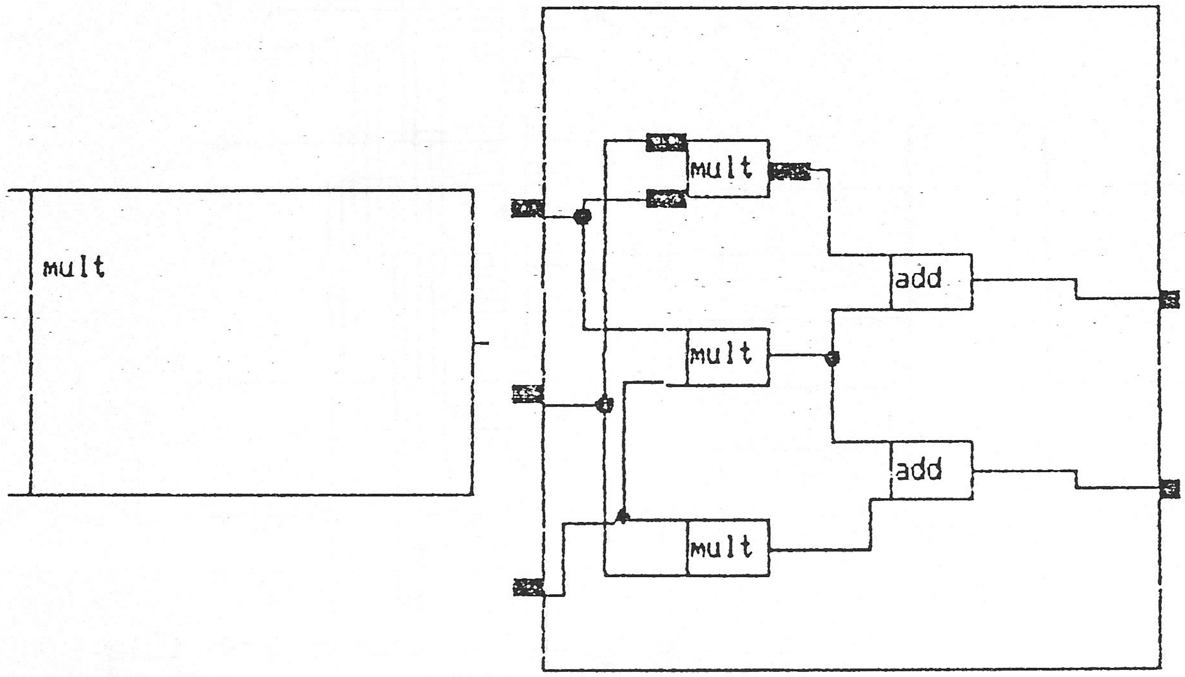

Figure 3: A Multiplier and its Board

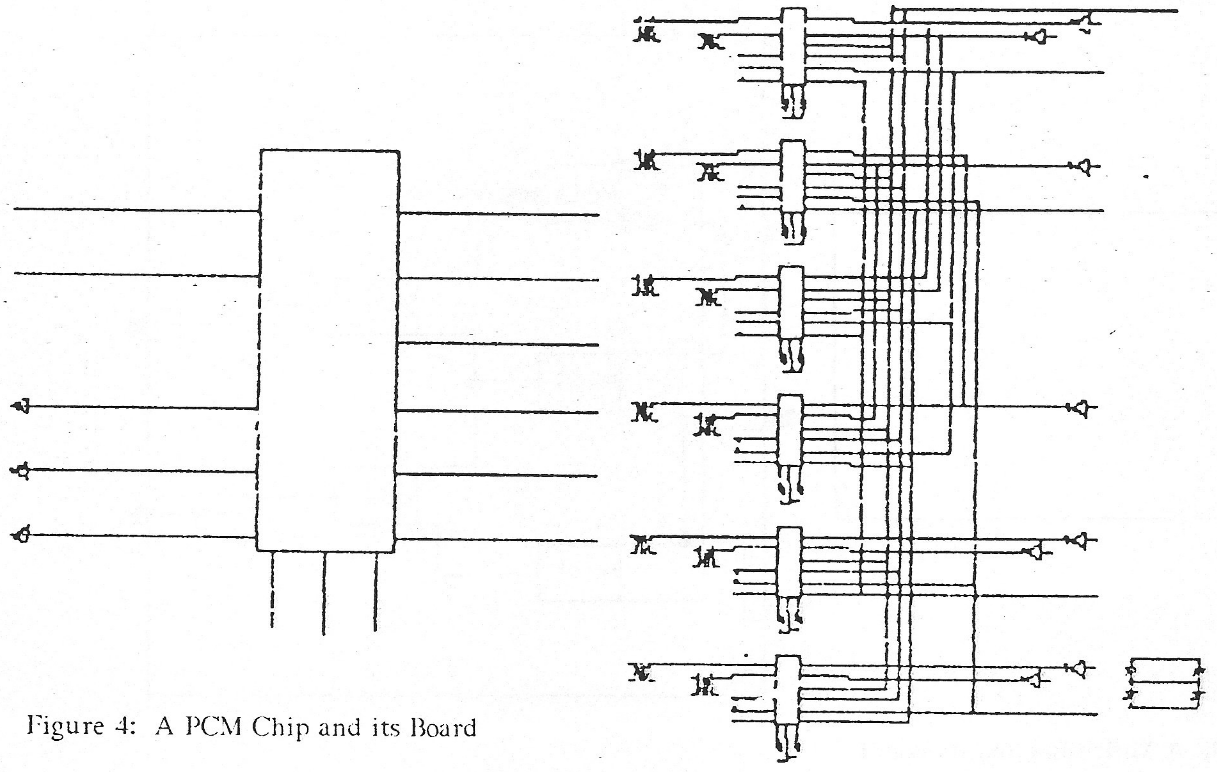

Figure 4: A PCM Chip and its Board

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. The mechanism of infertrace does of now not work for the PCM board for which we use a much more complicated representation system which has created unexpected interactions.

3. FUTURE WORK

Our future plans include the investigation of the knowledge representation scheme for display purposes. We also have noted interesting differences between structural and functional displays. These differences have to do with the different types of knowledge that have to be specified. While structural displays require considerably more fixed coordinate values, functional displays can replace this type of knowledge by knowledge about object clusters and their inner workings.

Part. III Co-lateral Educational Activities

1. Acquisition of Background Knowledge

Mr. Ming-Ruey Taie has passed his Ph. D. qualifying examination on "automatic fault diagnosis" at SUNY at Buffalo on March 1, 1985. Automatic fault diagnosis has got more and more attention of AI researchers recently due to the general shortage of qualified maintenance experts in a wide range of domains. Versatile fault diagnosis system for digital circuit boards is desirable due to the widespread using of the products and the fast introducing rate and relatively short market life of new devices. Various techniques used for fault diagnosis is surveyed, and the drawbacks of classical empirical-rule-based approach is discussed. A written version of the talk is available on request.

Mr. James Geller has passed his Ph. D. qualifying examination on "visual knowledge" at SUNY at Buffalo on May 3, 1985. A shortened written version of this talk is available on request. The field of visual knowledge is not very accepted as a separate field of AI knowledge. Only recently the interest in treating visual knowledge as a unified field has been growing [2]. For an earlier attempt to deal with this field see e.g. [13]. Because of this missing focus in the research literature a wide range of sources had to be used. Literature related to visual knowledge can be found in many sub areas of computer science, like data base mangement for pictorial databases [20] higher level computer vision [11] computer graphics [7] and combined language/graphics or language/vision interfaces [19]. Relevant work has been done as well in cognitive science and cognitive psycholgy [8] and deals with reasoning based on analogical representations. It was also necessary to study some of the classical literature on knowledge representation like [1,10] and many more.

The acquisition of domain related knowledge for the maintenance aspect of the problem was done by studying related conference proceedings and the SIGART Special Issue on Maintenance, containing papers like [15].

IJCAI-85 was attended by Drs. Shapiro and Srihari and Mr. Geller, where talks by authors in the maintenance field were visited, e.g. Bob Milne [12].

An interview with a local domain expert was conducted by Dr. Shapiro and Mr. Geller, however only small amounts of new knowledge about expert behavior have been contributed by this activity.

2. Unsupported Participants

Over a period of one semester one unsupported undergraduate student was involved in recoding a simple CAD like package that is used as the basis for much of the work done in the graphics interface part of the project. Currently two undergraduate students are working to extend our graphical routines to our Raster Technology/10 display device and to update a package for our Grinnell which was initially written and not maintained because of temporary Grinnell hardware problems.

3. Audio Visual Technology

We have investigated and tested methods for creating 35 mm slides and plotted transparencies for presentations. This investigation has been based on the DI-3000 device independent graphics system, a techniques so far not used in our Dept.; no application of these techniques for the VMES project can be reported.

References

1. Ronald J. Brachman and James G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, ().
2. Randall Davis, Howard Shrobe, and al., "The Hardware Troubleshooting Group," *SIGART Newsletter* **93**(Jul. 1985).
3. R. Davis, "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence* **24** pp. 347-410 (1984).
4. M. R. Genesereth, "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence* **24** pp. 411-436 (1984).
5. R. T. Hartley, "CRIB: Computer Fault-finding Through Knowledge Engineering," *Computer*, (March 1984).
6. Gary G. Hendrix, "Encoding Knowledge in Partitioned Networks," pp. 51-92 in *Associative Networks: Representation and Use of Knowledge by Computers*, ed. Nicholas Findler, (1979).
7. E. C. Kingsley, N. A. Schofield, and K. Case, "SAMMIE - A Computer Aid for Man Machine Modeling," *Computer Graphics* **15**(3)(Aug. 1981).
8. Stephen M. Kosslyn and Steven P. Shwartz, "A Simulation of Visual Imagery," *Cognitive Science* **1** p. 265 (1977).
9. Andrew Latto, David Mumford, and Jayant Shah, *The Representation of Shape*, IEEE Workshop on Computer Vision Representation and Control (1984).
10. Hector J. Levesque, "Foundations of a Functional Approach to Knowledge Representation," *AI* **23**(1984).
11. Alan K. Mackworth, "On Reading Sketch Maps," *IJCAI - 77*, (1977).
12. Robert Milne, "Fault Diagnosis through Responsibility," *IJCAI*, pp. 423-425 (1985).
13. Fanya S. Montalvo, "Visual Knowledge Representation and Acquisition," *Second Annual Meeting of the Cognitive Science Conference*, p. Session #12 (1980).
14. Joyce K. Reynolds, Jonathan B. Postel, Alan R. Katz, Gregg G. Finn, and Annette L. DeSchon, "The DARPA experimental multi media mail system," *Computer*, pp. 82-91 (Oct. 1985).
15. Ethan A. Scarl, John R. Jamieson, and Carl I. Delaune, "Process Monitoring and Fault Location at the Kennedy Space Center," *SIGART Newsletter* **93**(Jul. 1985).
16. 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).
17. S. C. Shapiro and The SNePS Implementation group, *SNePS User's Manual*, Department of Computer Science, SUNYAB, Buffalo, NY (revised: 1983).
18. Stuart C. Shapiro, Sargur N. Srihari, James Geller, Ming-Ruey Taie, Chi Choy, and Albert Hanyong Yuhan, "A Graphics Interface to a rule-based system," SCEEE-PDP/84-30 SCEEE-PDP/84-31, Southeastern Center for Electrical Engineering Education, St. Cloud, FL 32769 (Jan. 1985).
19. E. H. Shortliffe, G. Adorni, A. Boccalatte, and M. DI Manzo, "Cognitive Models for Computer Vision," *COLING - 82*, American Elsevier/North Holland, (1982).
20. Hideyuki Tamura, "Image Database Management for Pattern Information Processing Studies," in *Pictorial Information Systems*, ed. K.S. Fu, ().