DIAGNOSIS BASED ON DESCRIPTION OF STRUCTURE AND FUNCTION

Randall Davis*, Howard Shrobe*, Walter Hamscher*

Kären Wieckert*, Mark Shirley*, Steve Polit**

* The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

** Digital Equipment Corp.

Abstract

While expert systems have traditionally been built using large collections of rules based on empirical associations, interest has grown recently in the use of systems that reason from representations of structure and function. Our work explores the use of such models in troubleshooting digital electronics.

We describe our work to date on (*i*) a language for describing structure, (*ii*) a language for describing function, and (*iii*) a set of principles for troubleshooting that uses the two descriptions to guide its investigation.

In discussing troubleshooting we show why the traditional approach --- test generation --- solves a different problem and we discuss a number of its practical shortcomings. We consider next the style of debugging known as violated expectations and demonstrate why it is a fundamental advance over traditional test generation. Further exploration of this approach, however, demonstrates that it is incapable of dealing with commonly known classes of faults. We explain the shortcoming as arising from the use of a fault model that is both implicit and inseparable from the basic troubleshooting methodology. We argue for the importance of fault models that are explicit, separated from the troubleshooting mechanism, and retractable in much the same sense that inferences are retracted in current systems.

Introduction

While expert systems have traditionally been built using large collections of rules based on empirical associations (e.g., [9]) interest has grown recently in the use of systems that reason from representations of structure and function (e.g., [8], [7], [5]). Our work explores the use of such models in troubleshooting digital electronics.

We view the task as a process of reasoning from behavior to structure, or more precisely, from misbehavior to structural defect. We are typically presented with a machine exhibiting some form of incorrect behavior and must infer the structural abberation that is producing it. The task is interesting and difficult because the devices we want to examine are complex and because there is no well developed theory of diagnosis for them.

Our ultimate goal is to provide a level of performance comparable to that of an experienced engineer, including reading and reasoning from schematics; selecting, running, and interpreting the results of diagnostics; selecting and interpreting the results of input test patterns, etc. The initial focus of our work has been to develop three elements that appear to be fundamental to all of these capabilities. We require (*i*) a language for describing structure, (*ii*) a language for describing function, and (*iii*) a set of principles for troubleshooting that uses the two descriptions to guide its investigation. This paper describes our progress to date on each of those elements.

In discussing troubleshooting we show why the traditional approach to reasoning about digital electronics --- test generation --- solves a different problem and we discuss a number of its practical shortcomings. We consider next the style of debugging known as violated expectations and demonstrate why it is a fundamental advance over traditional test generation. Further exploration of the violated expectation approach, however, demonstrates that it is incapable of dealing with commonly known classes of faults. We explain the shortcoming as arising from the use of a fault model that is both implicit and inseparable from the basic troubleshooting methodology. We argue for the importance of fault models that are explicit, separated from the troubleshooting mechanism, and retractable in much the same sense that inferences are retracted in current systems.

Structure Description

By structure description we mean topology --- the connectivity of components. A number of structure description languages have been developed, but most, having originated in work on machine design, deal exclusively with *functional* components, rarely making any provision for describing *physical* organization.[1] In doing machine diagnosis, however, we are dealing with a collection of hardware whose functional and physical organizations are both important. The same gate may be both (*i*) functionally a part of a multiplexor, which is functionally a part of a datapath, etc., and (*ii*) physically a part of chip E67, which is physically part of board 5, etc. Both of these hierarchies are relevant at different times in the diagnosis and both are included in our language.

We use the functional hierarchy as the primary organizing principle because, as noted, our basic task involves reasoning from function to structure rather than the other way around.[2] The functional organization is also typically richer than the structural (more levels to the hierarchy, more terms in the

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research on electronic troubleshooting is provided in part by the Digital Equipment Corporation.

1. This is curiously true even for languages billing themselves as *computer hardware description languages*. They rarely mention a piece of physical hardware.

2. We are typically confronted with a machine that misbehaves, not one that has visible structural damage.

vocabulary), and hence provides a useful organizing principle for the large number of individual physical components. Compare, for example, the functional organization of a board (e.g., a memory controller with cache, address translation hardware, etc.) with the physical organization (1 pc board, 137 chips).

The most basic level of our description vocabulary is built on three concepts: *modules*, *ports*, and *terminals* (Fig. 1). A module can be thought of as a standard black box. A module has at least two ports; ports are the place where information flows into or out of a module. Every port has at least two terminals, one terminal on the outside of the port and one or more inside. Terminals are primitive elements; they store logic levels representing the information flowing into or out of a device through their port, but are otherwise devoid of substructure.

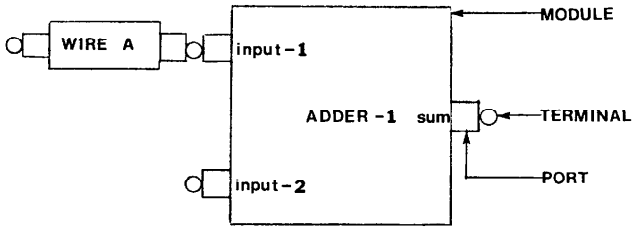

Figure 1 - The basic terms used in structure description.

Two modules are attached to one another by superimposing their terminals. In Fig. 1, for example, wire A is a module that has been attached to input-1 of the adder module in this fashion.

The language is hierarchical in the usual sense; modules at any level may have substructure. In practice, our descriptions terminate at the gate level in the functional hierarchy and the chip level in the physical hierarchy, since for our purposes these are black boxes --- only their behavior (or misbehavior) matters. Fig. 2 shows the next level of structure of the adder and illustrates why ports may have multiple terminals on their inside: ports provide the important function of shifting level of abstraction. It may be useful to think of the information flowing along wire A as an integer between 0 and 15, yet we need to be able to map those four bits into the four single-bit lines insider the adder. Ports are the place where such information is kept. They have machinery (described below) that allows them to map information arriving at their outer terminal onto their inner terminals. The default provided in the system accomplishes the simple map required in Fig. 2.

Since our ultimate intent is to deal with hardware on the scale of a mainframe computer, we need terms in the vocabulary capable of describing levels of organization more substantial than the terms used at the circuit level. We can, for example, refer to *horizontal*, *vertical*, and *bitslice* organizations, describing a memory, for instance, as "two rows of five 1K ram's". We use these specifications in two ways: as a description of the organization of the device and a specification for the pattern of interconnections among the components.

Our eventual aim is to provide an integrated set of descriptions that span the levels of hardware organization ranging from interconnection of individual modules, through higher level of organization of modules, and eventually on up through the register transfer and PMS level [2]. Some of this requires inventing vocabulary like that above, in other places (e.g., PMS)

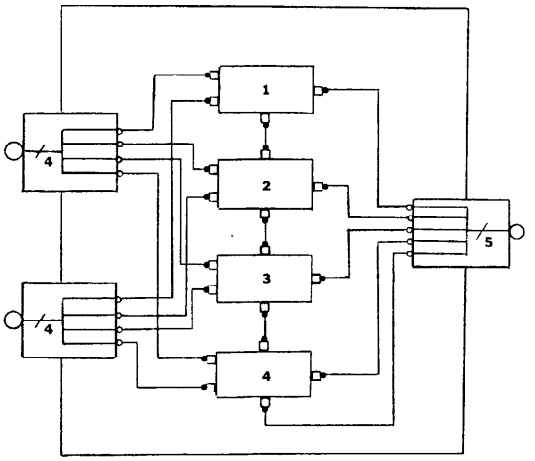

Figure 2 - Next level of structue of the adder.

we may able to make use of existing terminology and concepts.

The structural description of a module is expressed as a set of commands for building the module. Hence the adder of Fig. 2 is described by indicating how to "build" it (Fig. 3). These commands are then executed by the system, causing it to build data structures that model all the components and connections shown. The resulting data structures are organized around the individual components. Executing the first expression of Fig. 3, for example, produces 4 data structures that model the individual slices of the adder.

```
(definemodule  adder
  (repeat 4 i
       (part slice-i adder-slice)
       (run-wire (input-1 adder) (input-1 slice-i))
       (run-wire (input-2 adder) (input-2 slice-i))
       (run-wire (output slice-i)  (sum adder))

  (repeat 3 i
       (run-wire (carry-out slice-i)
                 (carry-in slice-[i+1])) ))
```

Figure 3 - Parts are described by a pathname through the part hierarchy, e.g., (input-1 adder). (This description can be abbreviated as a *bitslice* organization, but is expanded here for illustration.)

This approach to structure description offers two interesting properties: (a) a natural merging of procedural and object-oriented descriptions, and (b) the use of analogic representations.

To see the merging of descriptions, note that we have two different ways of thinking about structure. We describe a device by indicating how to build it (the procedural view), but then want to think about it as a collection of individual objects (the object-oriented view). The first view is convenient for *describing* structure, the second makes it easy to *answer questions* about it, questions like connectivity, location, etc., that are important in signal tracing and other troubleshooting techniques. The two descriptions are unified because the system simply "runs" the procedural description to produce the data structures modeling

the device. This gives us the benefit of both approaches with no additional effort and no chance that the two will get out of sync.

The representation is analogic because the data structures that are built are isomorphic to the structure being described. "Superimposing" two terminals, for instance, is implemented as a merging of the structure representing the terminals. The resulting data structures are thus connected in the LISP sense in the same ways that the objects are connected in Fig. 2. The benefit here is primarily conceptual, it simply makes the resulting structures somewhat easier to understand.

Our description language has been built on a foundation provided by a subset of DPL [1]. While DPL as originally implemented was specific to VLSI design, it proved relatively easy to "peel off" the top level of language (which dealt with chip layout) and rebuild on that base the new layers of language described above.

Since pictures are a fast, easy and natural way to describe structure, we have developed a simple circuit drawing system that permits interactive entry of pictures like those in Figs. 2 and 4. Circuits are entered with a combination of mouse movements and key strokes; the resulting structures are then "parsed" into the language shown in Fig. 3.

Behavior Description

A variety of techniques have been explored in describing behavior, including simple rules for mapping inputs to outputs, petri nets, and unrestricted chunks of code. Simple rules are useful where device behavior is uncomplicated, petri nets are useful where the focus is on modeling parallel events, and unrestricted code is often the last resort when more structured forms of expression prove too limited or awkward. Various combinations of these three have also been explored.

Our initial implementation uses constraints [10] to represent behavior. Conceptually a constraint is simply a relationship. The behavior of the adder of Fig. 1, for example, can be expressed by saying that the logic levels of the terminals on ports *input-1*, *input-2* and *sum* are related in the obvious fashion. This is an expression of a relationship, not a commitment to a particular computation --- the logic level at any one of the terminals can be computed given the other two.

In practice, this is accomplished by defining a set of rules covering all different computations (the three for the adder are shown below) and setting them up as demons that watch the appropriate terminals. A complete description of a module, then, is composed of its structural description as outlined earlier and a behavior description in the form of rules that interrelate the logic levels at its terminals.

```
to get sum from (input-1 input-2) do (+ input-1 input-2)
to get input-1 from (sum input-2) do (- sum input-2)
to get input-2 from (sum input-1) do (- sum input-1)
```

A set of rules like these is in keeping with the original conception of constraints, which emphasized the non-directional, relationship character of the information. When we attempt to use it to model causality and function, however, we have to be careful. This approach is well suited to modeling causality and behavior in the world of analog circuits, where devices are largely non-directional. But we can hardly say that the last two rules above are a good description of the *behavior* of an adder chip --- the device doesn't do subtraction; putting logic levels at its output and one input does not cause a logic level to appear on its other input.

The last two rules really model the *inferences we make about the device*. Hence we find it useful to distinguish between rules representing *flow of electricity* (digital behavior, the first rule above) and rules representing *flow of inference* (conclusions we can make about the device, the next two rules). This not only keeps the representation "clean", but as we will see, it provides part of the foundation for the troubleshooting mechanism.

A set of constraints is a relatively simple mechanism for specifying behavior, in that it offers no obvious support for expressing behavior that falls outside the "relation between terminals" view. The approach also has known limits. For example, constraints work well when dealing with simple quantities like numbers or logic levels, but run into difficulties if it becomes necessary to work with symbolic expressions.[3]

The approach has, nevertheless, provided a good starting point for our work and offers two important advantages. First, the DPL and constraint machinery includes mechanisms for keeping track of dependency information --- an indication of how the system determined the value at a terminal --- expressed in terms of what rule computed the value and what other values the rule used in performing its computation. This is very useful in tracing backward to the source of the misbehavior.

Second, the system provides machinery for detecting and unwinding contradictions. A contradiction arises if two rules try to set different values for the same terminal. As we illustrate below, the combination of dependency information and the detection of contradictions provides a useful starting place for troubleshooting.

Our system design offers a number of features which, while not necessarily novel, do provide useful performance. For example, our approach offers a unity of device description and simulation, since the descriptions themselves are "runnable". That is, the behavior descriptions associated with a given module allow us to simulate the behavior of that module; the interconnection of modules specified in the structure description then causes results computed by one module to propagate to another. Thus we don't need a separate description or body of code as the basis for the simulation, we can simply "run" the description itself. This ensures that our description of a device and the machinery that simulates it can never disagree about what to do, as can be the case if the simulation is produced by a separately maintained body of code.

Our use of a hierarchic approach and the terminal, port, module vocabulary makes multi-level simulation very easy. In simulating any module we can either run the constraint associated with the terminals of that module (simulating the module in a single step), or "run the substructure" of that module, simulating the device according to its next level of structure. Since the abstraction shifting behavior of ports is also implemented with the constraint mechanism, we have a convenient uniformity and economy of machinery: we can enable either the constraint that spans the entire module or the constraint that spans the port.

Varying the level of simulation is useful for speed (no need to simulate verified substructure), and provides as well a simple check on structure and behavior specification: we can compare the results generated by the module's behavior specification with those generated by the next lower level of

3. What, for example, do we do if we know that the output of an or-gate is 1 but we don't know the value at either input? We can refrain from making any conclusion about the inputs, which makes the rules easy to write but misses some information. Or we can write a rule which express the value on one input in terms of the value on the other input. This captures the information but produces problems when trying to use the resulting symbolic expression elsewhere.

simulation. Mismatches typically mean a mistake in structure specification at the lower level.

We believe it is important in this undertaking to include descriptions of both design and implementation, and to distinguish carefully between them. A wire, for example, is a device whose behavior is specified simply as the guarantee that a logic level imposed on one of its terminals will be propagated to the other terminal. Our structure description allows us to indicate the *intended* direction of information flow along a wire, but our simulation is not misled by this. This is, of course, important in troubleshooting, since some of the more difficult faults to locate are those that cause devices to behave not as we know they "should", but as they are in fact electrically capable of doing. Our representation machinery allows us to include both design specifications (the functional hierarchy) and implementation (the physical hierarchy) and keep them distinct.

Finally, the behavior description is also a convenient mechanism for fault insertion. A wire stuck at zero, for example, is modeled by giving the wire a behavior specification that maintains its terminals at logic level 0 despite any attempt to change them. Bridges, opens, etc., are similarly easily modeled.

Troubleshooting

The traditional approach to troubleshooting digital circuitry (e.g., [3]) has, for our purposes, a number of significant drawbacks. Perhaps most important, it is a theory of *test generation*, not a theory of *diagnosis*. Given a specified fault, it is capable of determining a set of input values that will detect the fault (ie, a set of values for which the output of the faulted circuit differs from the output of a good circuit). The theory tells us how to move from faults to sets of inputs; it provides little help in determining what fault to consider, or which component to suspect.

These questions are a central issue in our work for several reasons. First, the level of complexity we want to deal with precludes the use of diagnosis trees, which can require exhaustive consideration of possible faults. Second, our basic task is repair, rather than initial testing. Hence the problem confronting us is "Given the following piece of misbehavior, determine the fault." We are not asking whether a machine is free of faults, we know that it fails and know how it fails. Given the complexity of the device, it is important to be able to use this information as a focus for further exploration.

A second drawback of the existing theory is its use of a set of explicitly enumerated faults. Since the theory is based on boolean logic, it is strongly oriented toward faults whose behavior can be modeled as some form of permanent binary value, typically the result of stuck-ats and opens. One consequence of this is the paucity of useful results concerning bridging faults.

A response to these problems has been the use of what we may call the "violated expectation" approach ([6], [4], [7]). The basic insight of the technique is the substitution of violated expectations for specific fault models. That is, instead of postulating a possible fault and exploring its consequences, the technique simply looks for mismatches between the values it expected from correct operation and those actually obtained. This allows detection of a wide range of faults because misbehavior is now simply defined as anything that isn't correct, rather than only those things produced by a struck-at on a line.

This approach has a number of advantages. It is, first of all, fundamentally a diagnostic technique, since it allows systematic isolation of the possibly faulty devices, and does so without having to precompute fault dictionaries, diagnosis trees, or the like. Second, it appears to make it unnecessary to to specify a set of expected faults (we comment further on this below). As a result, it can detect a much wider range of faults, including any systematic misbehavior exhibited by a single component. The approach also allows natural use of hierarchical descriptions, a marked advantage for dealing with complex structures.

This approach is a good starting point, but has a number of important limitations built into it. We work through a simple example to show the basic idea and use the same example to comment on its shortcomings.

Consider the circuit in Fig. 4.[4] If we set the inputs as shown, the behavior descriptions will indicate that we should expect 12 at F. If, upon measuring, we find the value at F to be 10, we have a conflict between observed results and our model of correct behavior. We check the dependency record at F to find that the value expected there was determined using the behavior rule for the adder and the values emerging from the first and second multiplier. One of those three must be the source of the conflict, so we have three hypotheses: either the adder behavior rule is inappropriate (ie, the first adder is broken), or one of the two inputs did not have the expected values (and the problem lies further back).

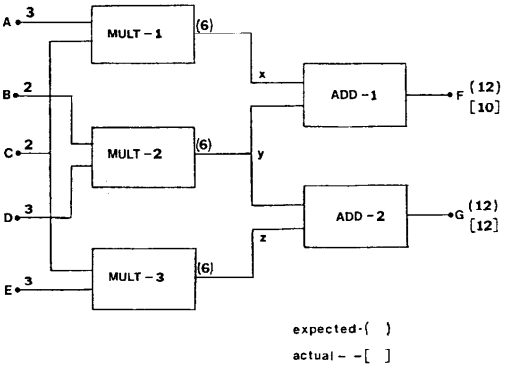

Figure 4 - Troubleshooting example using violated expectations.

If the second input to adder-1 was good, then the first input must have been a 4 (reasoning from the result at F, valid behavior of the adder, and one of the inputs). But that conflicts with our expectation that it should be a 6. That expectation was based on the behavior rule for the multiplier and the expected value of its inputs. Since the inputs to the multiplier are primitive (supplied by the user), the only alternative along this line of reasoning is that the multiplier is broken. Hence hypothesis # 2 is that adder-1 is good and multiplier-1 is faulty.

If the first input to adder-1 is good, then the second input must have been a 4 (suggesting that the second multiplier might be bad). But if that were a 4, then the expected value at G would be 10 (reasoning forward through the second adder). We can check this and discover in this case that the output at G is 12. Hence the value on the output of the second multiplier can't be 4,

4. As is common in the field, we make the usual assumptions that there is only a single source of error and the error is not transient. Both of these are important in the reasoning that follows.

it must be 6, hence the second multiplier can't be causing the current problem.

So we are left with the hypotheses that the malfunction lies in either the first multiplier or the first adder. The diagnosis proceeds in this style, dropping down levels of structural detail as we begin to isolate the source of the error.

This approach is a useful beginning, but has some clear shortcomings that result from hidden assumptions about faults. Consider the slightly revised example shown in Fig. 5. Reasoning just as before,[5] the fault at F leads us to suspect adder-1. But if adder-1 is faulty, then everything else is good. This implies a 6 on lines y and z, and (reasoning forward) a 12 at G. But G has been measured to be 6, hence adder-1 can't be responsible for the current set of symptoms. If adder-1 is good, then the fault at F might result from bad inputs (lines x and y). If the fault is on x, then y has a 6. But (reasoning forward) this means a 12 at G. Once again we encounter a contradiction and eliminate line x as a candidate. We turn to line y, postulate that it is 0. This is consistent with the faults at both F and G, and is in fact the only hypothesis we can generate.

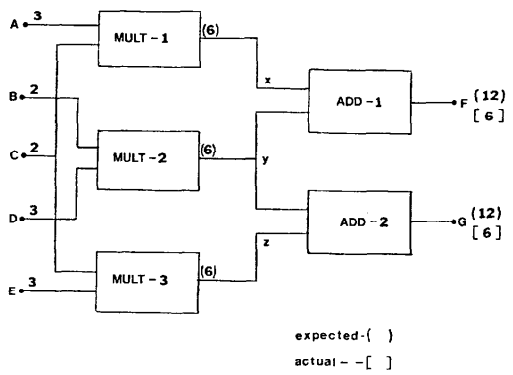

Figure 5 - Troublesome troubleshooting example.

The key phrase here is "the only hypothesis we can generate". In fact, there is another quite reasonable hypothesis: the third multiplier might be bad.[6] But how could this produce errors at both F and G? The key lies in being wary of our models. The thought that digital devices have input and output ports is a convenient abstraction, not an electrical reality. If, as sometimes happens (due to a bent pin, bad socket, etc.), a chip fails to get power, its inputs are no longer guaranteed to act unidirectionally as inputs. If the third multiplier were a chip that failed to get power, it might not only send out a 0 along wire z, but it might also pull down wire C to 0. Hence the symptoms result from a single point of failure (multiplier-3), but the error propagates along an "input" line common to two devices.

The problem with the traditional violated expectation approach lies in its implicit acceptance of unidirectional ports and the reflection of that acceptance in the basic dependency-unwinding machinery. That machinery implicitly believes that inputs only get information from outputs --- when checking the inputs to multiplier-1, we said they were "primitive". We looked only at the input terminals A and C, never at the other end of the wire at multiplier-3.

Bridges are a second common fault that illustrates an interesting shortcoming in the contradiction detection approach. The reasoning style used above can never hypothesize a bridging fault, again because of implicit assumptions about the model and their subtle reflection in the method. Bridges can be viewed as wires that don't show up in the design. But the traditional approach makes an implicit "closed world" assumption --- the structure description is assumed to be complete and anything not shown there "doesn't exist". Clearly this is not always true. Bridges are only one manifestation; wiring errors during assembly are another possibility.

Let's review for a moment. One problem with the traditional test generation technology was its use of a very limited fault model. The contradiction detection approach improves on this substantially by defining a fault as anything that produces behavior different from that expected. This seems to be perfectly general, but, as we illustrated, it is in fact limited in some important ways.

So what do we do? If we toss out the assumption that input and output ports are unidirectional, we take care of that class of errors; the cost is generating more hypotheses. Perhaps we can deal with the increase. If we toss out the closed-world assumption and admit bridges, we're in big trouble. Even if we switch to our physical representation[7] to keep the hypotheses constrained to those that are physically plausible, the number is vast. If we toss out the assumption that the device was wired as the description indicates, we're in big trouble even if we invoke the single point of failure constraint and assume only one such error But some failures are due to multiple errors... and transients are an important class of errors ... and .... Wait, down this road appears to lie madness, or at the very least, chaos.

What can we do? We believe that the important thing to do is what human experts seem to do:

> *Make all the simplifying assumptions we have to to keep the problem tractable.*
> *Be explicitly aware of what those assumptions are.*
> *Be aware of the effect the assumptions have on candidate generation and testing.*
> *Be able to discard each assumption in turn if it proves to be misleading.*

The key, it seems, lies in determining what are the appropriate layers of assumptions for a given domain and in determining their effects on the diagnostic process. In our domain, for example, a sample list of the assumptions underlying correct function of a circuit might be:

no wires are stuck
no wires present other than those shown
ports functioning in specified direction
actual assembly matches design specifications
original design is correct

Surrendering these one by one leads us to consider stuck-ats, then bridges, then power loss, etc. We have significant work yet to do in determining a more complete and correct list, and in determining the consequences of each assumption on the diagnostic process. But we feel this is a key to creating more interesting and powerful diagnostic reasoners.

5. The eager reader has no doubt already chosen a likely hypothesis. We go through the reasoning in any case, to show that the method outlined generates the same hypothesis and is in fact simply a more formal way of doing what we often do intuitively.

6. Or the first.

7. Remember, we said it was important to have one.

References

[1] Batali J, Hartheimer A, The design procedure language manual, MIT AI Memo 598, Sept. 1980.

[2] Bell G, Newell A, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[3] Breuer M, Friedman A, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.

[4] Brown J S, Burton R, deKleer J, Pedagogical and knowledge engineering techniques in the SOPHIE systems, Xerox Report CIS-14, 1981.

[5] Davis R, Expert systems: Where are we and where do we go from here? *AAAI Magazine*, summer 1982.

[6] deKleer J, Local methods for localizing faults in electronic circuits, MIT AI Memo 394, 1976.

[7] Genesereth M, The use of hierarchical models in the automated diagnosis of computer systems, Stanford HPP memo 81-20, December 1981.

[8] Patil R, Szolovits P, Schwartz W, Causal understanding of patient illness in medical diagnosis, *Proc. IJCAI-81*, August 1981, pp 893 899.

[9] Shortliffe E, *Computer-Based Medical Consultations: Mycin*, American Elsevier, 1976.

[10] Sussman G, Steele G, Constraints - a language for expressing almost-hierarchical descriptions, *AI Journal*, Vol 14, August 1980, pp 1-40.