MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I.MEMO 739 JUNE, 1984

DIAGNOSTIC REASONING BASED ON STRUCTURE AND BEHAVIOR

Randall Davis
The Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

We describe a system that reasons from first principles, i.e., using knowledge of structure and behavior. The system has been implemented and tested on several examples in the domain of troubleshooting digital electronic circuits. We give an example of the system in operation, illustrating that this approach provides several advantages, including a significant degree of device independence, the ability to constrain the hypotheses it considers at the outset, yet deal with a progressively wider range of problems, and the ability to deal with situations that are novel in the sense that their outward manifestations may not have been encountered previously.

As background we review our basic approach to describing structure and behavior, then explore some of the technologies used previously in troubleshooting. Difficulties encountered there lead us to a number of new contributions, four of which make up the central focus of this paper.

We describe a technique we call *constraint suspension* that provides a powerful tool for troubleshooting.

We point out the importance of making explicit the assumptions underlying reasoning and describe a technique that helps enumerate assumptions methodically.

The result is an overall strategy for troubleshooting based on the progressive relaxation of underlying assumptions. The system can focus its efforts initially, yet will methodically expand its focus to include a broad range of faults.

Finally, abstracting from our examples, we find that the concept of *adjacency* proves to be useful in understanding why some faults are especially difficult and why multiple different representations are useful.

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 and in part by the Defense Advanced Research Projects Agency.

© Massachusetts Institute of Technology 1984

CONTENTS

1. INTRODUCTION 3
2. CENTRAL CONCERNS 5
3. DESCRIBING STRUCTURE 7
4. DESCRIBING BEHAVIOR 11
    4.1 Simulation Rules and Inference Rules 11
    4.2 Implementation 11

5. TROUBLESHOOTING 13
    5.1 The Traditional Approach 13

6. DISCREPANCY DETECTION AND CANDIDATE GENERATION 15
    6.1 Constraint Suspension 16
    6.2 Advantages of Discrepancy Detection and Constraint Suspension 19
    6.3 Subtleties in Candidate Generation 20

7. MECHANISM AND KNOWLEDGE 22
8. ORGANIZING THE PATHWAYS OF INTERACTION 23
9. EXAMPLE: A BRIDGE FAULT 25
    9.1 The Example 26
    9.2 The Example: Summary and Comments 31

10. CATEGORIES OF FAILURE 33
    10.1 Origins 33
    10.2 Ordering 34

11. THE ADJACENCY PRINCIPLE 36
12. EVALUATION AND FUTURE WORK 39
    12.1 Advantages of Reasoning From Structure and Behavior 39
    12.2 Describing Structure and Behavior 40
    12.3 Limitations in Candidate Generation 43
    12.4 Comments on the Example 43
    12.5 The Example: Implementation 44
    12.6 Scaling: Device Complexity and Time 44
    12.7 Limits of Modeling: Analog Devices and Incomplete Models 45
    12.8 But That's Not How It's *Really* Done 46

13. RELATED WORK 47
    13.1 Hardware Diagnosis 47
    13.2 Fault Models and Categories of Failure 48
    13.3 Other AI Work 48

14. SUMMARY 51

1. INTRODUCTION

The overall goal of this research is to develop a theory of reasoning that exploits knowledge of structure and behavior. We proceed by building programs that use such knowledge to reason from first principles in solving problems. The initial focus is troubleshooting digital electronic hardware, where we have implemented a system based on a number of new ideas and tools.

Troubleshooting digital electronics is a good domain for several reasons. First, troubleshooting seems to be one good test of part of what it means to "understand" a device. We view the task as a process of reasoning from behavior to structure, or more precisely, from misbehavior to structural defect: given symptoms of misbehavior, we are to determine the structural abberation responsible for the symptoms. Second, the task is interesting and difficult because the devices are complex and because there is no established theory of diagnosis for them. Third, the domain is appropriate because the required knowledge is readily available from schematics and manuals. Finally, the application itself is relevant and tractable.

Work with a similar intent has been done in other domains, including medicine [23], computer-aided instruction [6], and electronic troubleshooting [7,18], with the "devices" ranging from the gastro-intestinal tract, to transistors and digital logic components.

This work is novel in a number of respects, some of which have been reported in an earlier publication [8]. As noted there:

> We have developed languages that distinguish carefully between structure and behavior, and that provide multiple descriptions of structure, organizing it both functionally and physically.
>
> We have argued that the concept of *paths of causal interaction* is a primary component of the knowledge needed to do reasoning from structure and behavior.
>
> We have developed an approach to troubleshooting based on the use of a layered set of categories of failure and demonstrated its use in diagnosing a bridge fault.

This paper substantially expands and develops this line of work,[1] reporting several new contributions:

> We describe a new technique called *constraint suspension*, capable of determining which components can be responsible for an observed set of symptoms.
>
> We show that the categories of failure, previously derived informally, can be given a systematic foundation. We show that the categories can be generated by examining the assumptions underlying our representation.
>
> This in turn gives us a new way to view our approach to troubleshooting: the methodical enumeration and relaxation of assumptions about the device.
>
> Finally, we describe the concept of *adjacency* and argue that it helps in understanding both what it means to have a good representation, and why multiple representations can be useful.

Section 2 is introductory, supplying a brief review of the overall concerns, expanding on the ideas listed above as a preview of what is central to this work.

---

1. For the sake of continuity and ease of presentation, parts of [8] are reprinted here, including the bridge fault example and some of the background material (machinery for describing structure and function, and the introduction to troubleshooting). This material is reprinted with permission from [8], Copyright Academic Press Inc. (London) Ltd.

If we are to reason from first principles, we require: (*i*) a language for describing structure, (*ii*) a language for describing behavior, and (*iii*) a set of principles for troubleshooting that uses the two descriptions to guide its investigation. The central part of the paper describes each of those components, paying particular attention to the nature of the reasoning that underlies troubleshooting.

Sections 3 and 4 explore our approach to describing structure and behavior, while Sections 5 through 9 deal with troubleshooting. In Section 5 we consider the machinery traditionally used to reason about circuits and show that it fails to solve the problem we face. We then show how a technique called discrepancy detection offers a number of useful advances, but claim that it must be used with care. We suggest that the potential difficulties center around implicit assumptions typically made when using the technique. We find a solution to the problems encountered by making those assumptions explicit, organizing them appropriately, and providing for a way to surrender them one by one, methodically expanding the scope of failure the program considers. Section 9 provides an example of these ideas in operation, showing how they guide the diagnosis of a bridge fault.

In Sections 10 and 11 we draw back from the specific example presented and attempt to generalize our methods. We argue that the concept of paths of interaction offers a useful framework for asking questions about a domain. We claim that the notion of "adjacency" both explains some of the difficulty in reasoning about bridges and provides guidance in selecting and using multiple representations.

Section 12 reviews the machinery we have presented, pointing out its limitations and suggesting directions for future work. Finally, Section 13 compares our work to a number of previous approaches to similar problems.

2. CENTRAL CONCERNS

Some of the examples in later sections of this paper require a substantial amount of detail, yet the principles they illustrate are often relatively uncomplicated. In order to be sure that the important points are not lost in the detail, we describe the fundamental principles here briefly, enlarging on them in the remainder of the paper.

† *Candidate generation can be achieved by using a new technique we call constraint suspension.*

Given the symptoms of a malfunction, we wish to generate candidate components, i.e., determine which components could plausibly be responsible for the malfunction. We model the intended behavior of a device as a network of interconnected constraints, where each constraint models the behavior of one of the components. If the device is malfunctioning, then the outputs predicted by the entire constraint network will not match the actual outputs. Normally, contradictions in constraint networks are handled by retracting one or more of the input values. Constraint suspension takes the dual view and asks instead, *Is there some constraint (component behavior) whose retraction will leave the network in a consistent state?* Each such constraint that we find corresponds to a component whose misbehavior can account for all the observed symptoms. In many cases the technique will also supply the details of the component misbehavior (i.e., show how it is misbehaving). We describe the technique in detail in Section 6.1.

† *Paths of causal interaction play a central role in troubleshooting.*

An important part of the basic knowledge needed for troubleshooting is understanding the mechanisms and pathways by which one component can affect another. Electronic components typically interact because they are wired together, but they can also interact due to heat, capacitive coupling, etc. We argue that viewing the problem in terms of paths of interaction is more useful and revealing than the fault models (e.g., stuck-ats) traditionally used.

† *In doing troubleshooting we are faced with an unavoidable problem of complexity vs. completeness.*

To be good at troubleshooting, we need to handle many different kinds of paths of interaction. But this presents a serious problem. If we include all possible paths, candidate generation becomes indiscriminate: every component could somehow be responsible for the observed symptoms. Yet omitting any one path would make it impossible to diagnose an entire class of faults.

† *One technique for dealing with this dilemma is enumerating and layering the categories of failure.*

As experienced troubleshooters know, some things are more likely to go wrong than others. But what are the "some things"? And what does "more likely" mean? We make both of these notions more precise.

We show how to generate the "things that can go wrong" by making explicit the assumptions underlying our representation; the result is a list of "categories of failure". The categories are organized by using the most likely first and adding additional, less likely categories only in the face of contradictions.

Associated with each category of failure is a collection of paths of interaction. Hence the ordered listing of categories produces an ordering on the paths of interaction to be considered. This allows us to constrain the paths we consider initially, making it possible to constrain candidate generation. But no path is permanently excluded, hence no class of faults is overlooked.

† *The result is a strategy for troubleshooting based on the methodical*

enumeration and relaxation of underlying assumptions.
This technique allows us to deal with a progressively wider range of different faults without being overwhelmed by too many candidates at any one step.

† *The categories of failure can be generated systematically by examining the assumptions underlying our representation.*

As we demonstrate in Section 10, the categories can be derived by listing the assumptions implicit in the representation and considering the consequences of violating each in turn. Because the representation employs little more than the traditional notion of black boxes, the categories of failure deal with information transmission, and the results of this exercise have relevance broader than digital electronics. We suggest that the results are applicable to software and speculate about wider applications.

† *The concept of <u>adjacency</u> proves to be useful in both troubleshooting and the selection of representations.*

We noted above that the possible pathways for device interaction are an important part of the knowledge required for troubleshooting. We take the view that devices interact because they are in some sense *adjacent*: electrically adjacent (wired together), physically adjacent (hence "thermally connected"), electromagnetically adjacent (not shielded), etc. Each of these definitions can be used as the basis for a different representation of the device, different in its definition of adjacency. The multiplicity of possible definitions helps to explain why some faults are especially difficult to diagnose: *they result from interactions between components that are adjacent in a sense (representation) that is unusual or subtle.*

This concept appears to generalize in several ways. We view faults as modifications to the original design, and claim that changes that appear small and local in one representation may not appear small and local in another. We also suggest that adjacency can help determine what makes for a "good" representation: one in which the change can be seen as compact. It may also explain some of the utility of using multiple representations: they offer multiple different definitions of adjacency.

† *Adjacency is a useful principle to the extent that the <u>single initial cause heuristic</u> holds true.*

A common sense heuristic suggests that malfunction of a previously working device generally results from a single cause rather than a number of simultaneous, independent events. When there is indeed a single cause, there will be some representation in which a compact change accounts for the difference between the good and faulty device.

† *The concept of a <u>single point of failure</u> is not well defined without specification of the underlying representation.*

A failure in a single physical component, for example, may manifest as multiple points of failure in the behavior of the overall device. Consider a single chip with four AND gates in it. If the chip is damaged (via over-voltage, heat, mechanical stress, etc.), we will have a single point of failure in the physical representation, but may find four different points of failure (the four AND gates) in the functional view of the device. We claim as a result that "single point of failure" is by itself an under-determined concept. and that to make precise sense of the term we need to indicate the underlying representation to which "point" refers. We claim further that the use of multiple representations can add diagnostic power by offering different ways of resolving apparently independent failures into a single cause.

3. DESCRIBING STRUCTURE

If we wish to reason about structure and behavior, we clearly need a way of representing both. We consider each of these in turn.

By structure we mean information about the interconnection of modules. Roughly speaking, it is the information that would remain after removing all the textual annotation from a schematic.

Two different ways of organizing this information are particularly relevant to troubleshooting: functional and physical. The functional view gives us the machine organized according to how the modules interact; the physical view tells us how it is packaged. We thus prefer to replace the somewhat vague term "structure" by the slightly more precise terms *functional organization* and *physical organization*. As we will see, every device is described from both perspectives, producing two distinct, interconnected descriptions.

Our aim is to provide a means of encoding in one place all of the information about a circuit that is typically distributed across several different documents. To this end our approach provides a way of representing much of the information traditionally found in a schematic, as well as the hierarchical description often encoded in block diagrams.

The most basic level of our structure description is built on three concepts: *modules*, *ports*, and *terminals* (Fig. 1). A module can be thought of as a standard black box; 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 are the places we can "probe" to examine the information flowing into or out of a device through a port, but they are otherwise devoid of interesting substructure.

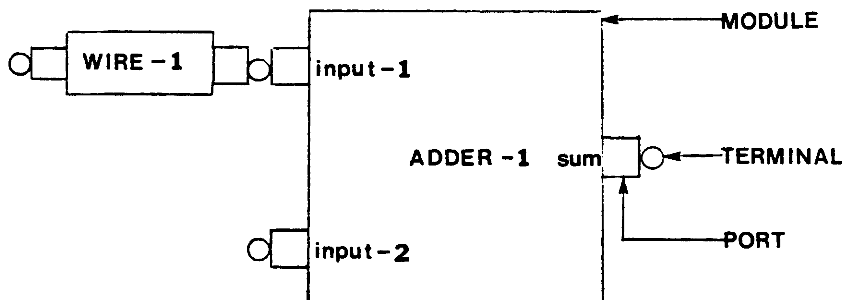

Figure 1 - The basic terms used in structure description.

Two modules are attached to one another by superimposing their terminals. In Fig. 1, for example, wire-1 is a (wire) module that has been attached to the input-1 port of adder-1 (an 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-1 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.

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 [4] levels. Some of this requires inventing vocabulary like that above, in other places we may able to make use of existing terminology and concepts.

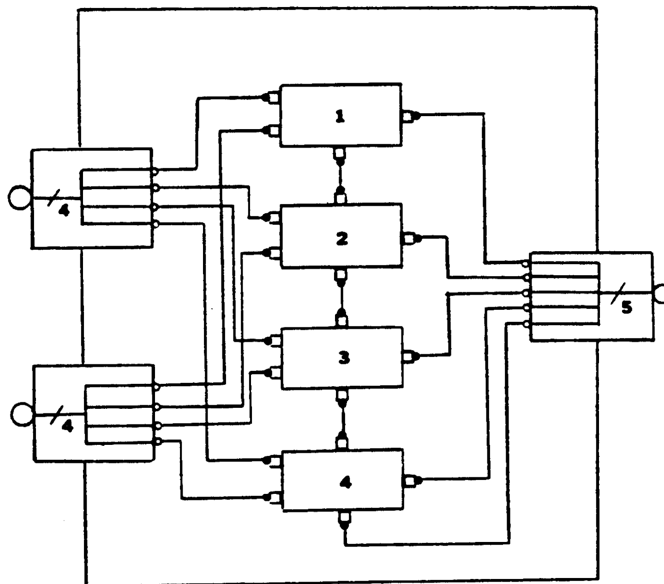

Figure 2 - Next level of structure of the adder, showing that it is implemented as a ripple-carry.

The structural description of a module is expressed as a set of commands for building it. The adder of Fig. 2, for example, is described by the instructions shown in Fig. 3. With `NBitsWide` bound to 4, the first expression indicates that we should repeat the following sequence of operations 4 times:

create an adder slice
run a wire from the first input of the adder to the first input of the slice
run a wire from the second input of the adder to the second input of the slice
run a wire from the output of the slice to the output of the adder

The next expression builds the carry chain and the last wires up the high order bit.

These commands are executed by the system, resulting in the creation of data structures that model all the components and connections shown. These data structures are isomorphic to the diagram shown above. That is, the data structures are connected in the LISP sense in the same ways that the objects are connected in Fig. 2. As in real devices, information flow occurs as a result of these interconnections: `slice-1` can place information on its end of the data structure modeling the carry-out wire, for example, because the two are superimposed. This information will then be propagated to the other end of the wire and pass into the data structure modeling `slice-2` because those two are superimposed.[2] The utility of this approach is explored below in Section 12.

```
(definemodule adder NBitsWide
    (repeat NBitsWide 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 (- NBitsWide 1) i  (run-wire (carry-out slice-i)
                                         (carry-in slice-[i+1])))
    (run-wire (carry-out slice-[NBitsWide - 1])  (sum adder)) )
```

Figure 3 - Parts are described by a pathname through the part hierarchy, e.g., (input-1 adder).

The definition in Fig. 3 is thus a specification of a prototypical ripple-carry adder; invoking it with a specific value for the width parameter produces one particular instance. We have found it useful to maintain this prototype/instance distinction for several reasons. It allows the standard economy of representation. since information common to all instances can be stored once with the prototype. It also makes possible the parameterization illustrated above, allowing us to describe the overall structure of the device, capturing the generalization inherent in the standard pattern of interconnections. Finally, it allows us to build up a library of module definitions available for later use. There is considerable utility in assembling such a library, utility beyond the ease of describing more complicated devices. By requiring that we define modules outside of the context of their use in any particular circuit, we encourage an important form of "mental hygiene" described further in Section 12.2.

Since our eventual aim is troubleshooting of devices as complex as a computer, the complete description of the device could become quite large. To deal with this we have made use of "lazy instantiation": when an instance of a module is created, only the "shell" is actually built at that time. For the adder above, for example, the outer "box" and ports are built, but the substructure --- the slices --- are not built until they are actually needed. Only when we need to "look inside" the box, i.e., drop down a level of structural detail, is additional structure actually built.

As a result, the system maintains a compact description of the device, expanding only where necessary. This can save a considerable amount of space. Even for the simple adder example above, there is a lot to describe: each slice is built from two half adders and an OR gate, each half-adder is built from an XOR and an AND gate. The initial instantiation is simply an adder with two inputs and an output, rather than four levels of detail and 20 gates. For more complex devices the savings can of course be greater.

The examples above illustrate how we describe functional organization: an adder composed of slices, which are in turn composed of half-adders, etc. Exactly the same module, port and terminal machinery is used to describe physical organization, but now the hierarchy of modules is cabinets, boards, and chips.

Since, as noted earlier, every component has both a functional and physical description, we have two interconnected description hierarchies. We represent this in our system by having the two hierarchies linked at their terminal nodes (Fig. 4).

2. We do not, by the way, have any special mechanism for dealing with wires. They are simply another module, albeit one with a particularly simple behavior: information presented to either port will be propagated to the other.

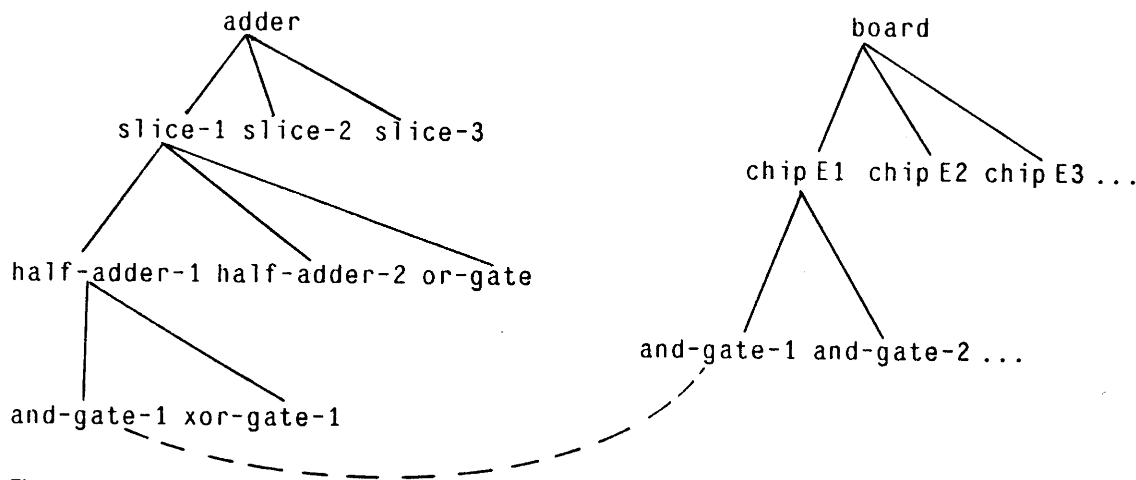

Figure 4 -- Interconnection of functional and physical descriptions

We determine the physical location of any non-terminal entry in the functional organization by aggregating the physical locations of all of its leaves. These descriptions are available at various levels of abstraction. The physical location of slice-1, for example, is the list of all the locations of its gates (e.g., E1, E4, E7, etc.); alternatively we can also say simply that it is on board-1, if all of those chips are in fact on the same board.

By having available cross-links between the two descriptions, it becomes possible to answer a range of useful questions. For example, we can ask, Where physically do I find the (part that functions as the) address translation register, or, What function(s) does this quad-and-gate chip perform?

Our description language has been built on a foundation provided by a subset of DPL [3]. 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. 1 and 2. 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.

4. DESCRIBING BEHAVIOR

By behavior we mean the black box description of a component: how is the information leaving the component related to the information that entered it? A variety of techniques have been used in the past to describe 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.

4.1 Simulation Rules and Inference Rules

Our initial implementation is based on a constraint-like approach ([30], [32]). 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.

In practice, this is accomplished by writing a set of expressions covering all the different individual relations (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 behavior of physical devices, however, we have to be careful. This approach is well suited to modeling behavior in 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 our implementation distinguishes between *simulation rules* that represent flow of electricity (digital behavior, the first rule above) and *inference rules* representing flow of inference (conclusions we can make about the device, the next two rules).

We find this distinction useful in part for reasons of "mental hygiene" (the two kinds of rules deal with different kinds of knowledge) and in part because it contributes to the robustness of the simulation in the face of unanticipated events. Consider for example a circuit that included an adder, and imagine that a fault in that circuit resulted in some other component trying to drive the output of the adder. If we treated all rules the same, our simulation would suggest that the device did the subtraction described above, yet this simply doesn't happen.

4.2 Implementation

Our implementation accomplishes this distinction by using two parallel but separate networks, one containing the simulation rules modeling causality, the other containing the inference rules. We keep them distinct by giving each terminal two slots, one holding the value computed by the simulation rules, the other holding the value computed by the inference rules (Fig. 5). Each

network then works independently, using the standard demon-like style of propagating values through its collection of slots.

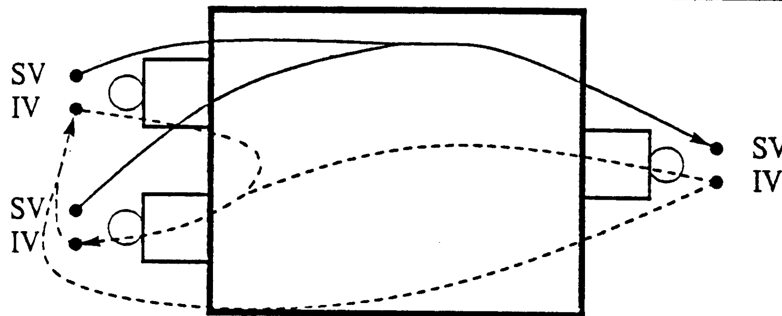

Figure 5 - Adder with one simulation rule (solid) and two inference rules (dashed). (SV - simulation value; IV - inference value)

In addition to firing rules to propagate information around the network, this rule-running machinery also keeps track of dependencies, allowing us to determine how the value in a slot got there. This is done by having each slot keep track of (i) every rule that uses this slot as an input, (ii) every rule that can place a value in this slot (i.e. uses it as an output) and (iii) which rule did, in fact, provide the current value (since, in general, more than one rule can set the value of a slot).[3]

This dependency network offers several advantages. As we explore below, it is one of the foundations of the discrepancy detection approach to troubleshooting. As the work in [6] demonstrated, such a network also makes it easy to explore hypothetical situations. We can place a value in some terminal and observe the consequences, i.e., see what values propagate from it. When done with the exploration, removing the value at the terminal causes the dependency network to remove everything that depends on it as well. This mechanism makes it easy to discover the answer to some questions by simulation: we can get the device to a particular state, then explore multiple alternative futures from that point.

While all of the examples given thusfar deal with combinatorial devices, we can also represent and model simple devices with memory. This requires three simple augmentations of the approach described above. First, we use a global clock and timestamp all values. Second, we extend the behavior rule vocabulary so that it can refer to *previous* values. For example, one simulation rule for a D flip-flop would be

```
    To get output from (input previous-output clock)
                      do (if clock is high then input else previous-input)
```

Finally, our propagation machinery is extended to keep a history of values at each terminal (details are in [10]). This model of time is still very simple, but has allowed us to represent and reason about basic flipflops, memories, etc.

3. The user can also set a value; this is most commonly done at the primary inputs.

5. TROUBLESHOOTING

Having provided a way of describing functional organization, physical organization and behavior, we come now to the important third step of providing a troubleshooting mechanism that works from those descriptions. We develop the topic in three stages. In the first stage we consider test generation --- the traditional approach to troubleshooting --- and explain how it falls short of our requirements.

We consider next the style of debugging known as *discrepancy detection* and demonstrate why it is a fundamental advance. Further exploration, however, demonstrates that this approach has to be used with care in dealing with some commonly known classes of faults. We suggest that the difficulties arise from a number of implicit assumptions typically made when troubleshooting.

In discussing how to deal with the difficulties uncovered, we argue for the primacy of *models of causal interaction*, rather than traditional fault models. We point out the importance of making these models explicit and separate from the troubleshooting mechanism. The result is a strategy for troubleshooting based on the systematic enumeration and relaxation of underlying assumptions. This approach allows us to deal with a progressively wider range of different faults, without being overwhelmed by too many candidates at any one step.

We demonstrate the power of this approach on a bridge fault, a traditionally difficult problem, showing how our system locates the fault in a focused process that generates only a few plausible candidates.

5.1 The Traditional Approach

The traditional approach to troubleshooting digital circuitry (e.g., [5]) relies primarily on the process of path sensitization in a range of forms, of which the D-Algorithm [26] is one of the most powerful. A simple example of path sensitization will illustrate the essential character of the process. Consider the circuit shown in Fig. 6 and imagine that we want to determine whether the wire labeled A is stuck at 1. We try to put a zero on it by setting both x1 and x2 to 0. Then, to observe the actual value on the wire we set x3 to 1, thereby propagating the value unchanged through G2, and set x4 to 0, making the output of G3 = 0, allowing the value of A to propagate unchanged through G1.

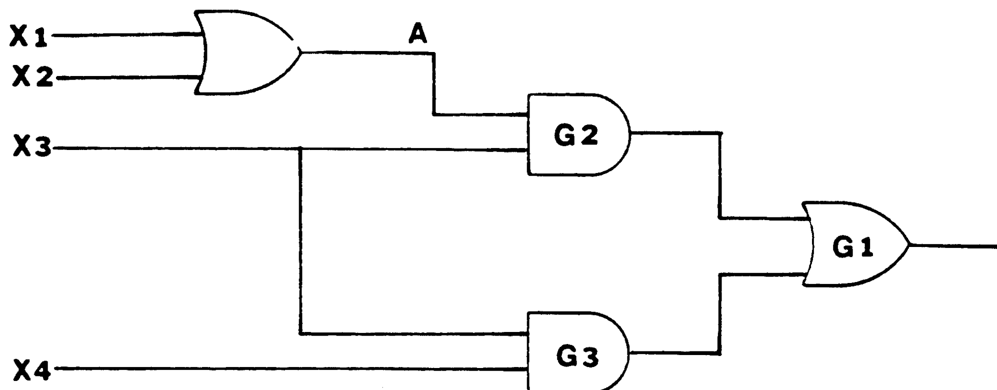

Figure 6 - Simple example of path sensitization.

Troubleshooting with this approach is then accomplished by running a complete set of such tests, checking for all stuck-ats on all wires.

For our purposes this approach has 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 (i.e., 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. Such questions are a central issue in our work because of complexity: the size of the devices we want to work with in the long run precludes the use of diagnosis trees --- complete decision trees for all possible faults.

A second drawback of the existing approach is its lack of sharp distinction between between *diagnosis* in the field and *verification* at the end of the manufacturing line. As a result of economic and historical forces, diagnostics written to verify the correct operation of a newly manufactured device have been pressed into service in the field. Yet the tasks are sharply different. We are not requesting verification that a machine is free of faults. The problem facing us is "Given the observed misbehavior, determine the cause." We know that the device has worked in the past, we know that some part of it has failed, and we know the symptoms of that failure. Given the complexity of the device, it is important to be able to use all of this information as a focus for further exploration. Only if this fails might it make sense to fall back on a set of diagnostics that, by design, start with no information and test exhaustively.

A final drawback of the existing theory is its unavoidable use of a set of explicitly enumerated fault models.[4] 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 results concerning bridging faults.[5]

In solving the diagnosis problem, though, we have a significant luxury: we can treat as an error any behavior that differs from the expected correct behavior. The misbehavior need not be modelable in terms of any fixed set of faults, it need only be different from what should have resulted.

In summary, the technology often used for troubleshooting is oriented toward test generation and the task of verification. We, on the other hand, are concerned with troubleshooting, a process that makes use of test generation (for an example, see [27]), but which requires as necessary precursors the processes of candidate generation (determining which components may be failing) and "symptom generation" (how they may be failing). The next section explores an approach that supports both of these.

---

4. Fault models are necessary in verification if we want to avoid the exponential effort of exhaustive testing. If we treat a device as a black box (e.g., saying only that it is an adder), we are forced to verify all of its behavior, a task that is potentially exponential in the number of inputs and amount of state. The most common way of avoiding this is by combining knowledge of the substructure of the device (e.g., that it is a carry-chain adder) with a specific set of faults to consider (e.g., stuck-ats on all wires), to produce tests for all such faults on all specified parts of the substructure. This task is at worst a product of the number of faults and wires.

5. While the theory underlying verification may be limited in the range of faults it can describe, in practice it turns out to handle a large part of the problem: experience indicates that a large percentage of all faults turn out to be *detected* (but not diagnosed) by checking just for stuck-ats. Hence we can determine that something is wrong (satisfying the *verification* task); determining the identity and location of the error (*diagnosis*), however, is a different problem.

6. DISCREPANCY DETECTION AND CANDIDATE GENERATION

One response to these problems has been the use of "discrepancy detection" (e.g., [11,6]). The two basic insights of the technique are (i) the substitution of violated expectations for specific fault models and (ii) the use of dependency records to trace back to the possible sources of a fault. Instead of postulating a possible fault and exploring its consequences, the technique 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.

The inspiration behind using dependency networks is that any component on the path from an input to an incorrect output could conceivably have been responsible for the faulty behavior. As we have seen, the simulator builds a dependency network by recording the propagation of values as it simulates the circuit. Using those records (or, equivalently, the original schematic) as a guide to which components to examine appears to be an effective way to focus attention appropriately. As we will see, it is in fact an interesting trap.

We work through a simple example to show the basic approach of discrepancy detection, then add to it the idea of constraint suspension. We comment on the strengths of the resulting procedure for candidate generation, then use the same example to illustrate difficulties that can arise.

Consider the circuit in Fig. 7.[6] If we set the inputs as shown, the system will use the behavior descriptions to simulate the circuit, constructing dependency records as it does so, and indicating that we should expect 12 at F.

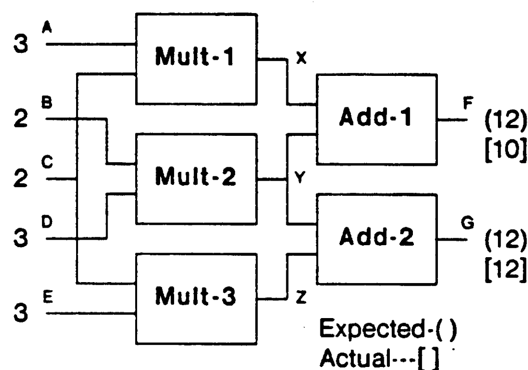

Figure 7 - Troubleshooting example using discrepancy detection.

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 trace back through the dependency network to enumerate the possible sources of the problem [11]. The dependency record at F indicates 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 possibilities to pursue: either the adder behavior rule is inappropriate (i.e., the first adder is broken),

6. 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 assumptions are important in the reasoning that follows; we comment further on them below.

or one of the two inputs did not have the expected values (and the problem lies further back). Consideration of the first possibility immediately generates hypothesis # 1: adder-1 is broken.

To pursue the second possibility, we assume that the second input to adder-1 is good. In that case 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 we expected a 6. Hence we now have a discrepancy at the input to adder-1; we have succeeded in pushing the discrepancy one step further back along the dependency chain. The expected value there 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.

Pursuing the third possibility: 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, it must be 6, so the second multiplier can't be causing the current problem.

This style of reasoning can be described as the *interaction of simulation and inference*: simulation generates expectations about correct outputs based on inputs and knowing how devices work (simulation rules); inference generates conclusions about actual behavior based on observed outputs and device inference rules. The comparison of these two, in particular differences between them, provides a foundation for troubleshooting.[7]

6.1 Constraint Suspension

In the discussion above, we glossed over the question of determining when a candidate is valid (as ADDER-1 was) and when it is inconsistent with all the available evidence (as MULT-2 was). One of the novel contributions of this work is the development of the constraint suspension technique as a way of providing an answer to this question.

To see how it works, consider once again the first step of the problem. When we examine the dependency record at F, we find that the value there resulted from the behavior rule for ADDER-1 and the values coming from MULT-1 and MULT-2. As above, the first possibility is that ADDER-1 is broken.

But this is a *local inference* (i.e., it is based only on the dependency record at F) and we have to be sure it's *globally consistent* with all the symptom data. More precisely, we want to ask whether there is *some* assignment of values to the ports of the adder that is consistent with all of the inputs and observed outputs. Is there any way in which ADDER-1 alone could be broken and produce the symptoms noted?

We can do this conveniently by using the "constraint-like" character of our representation and the notion of *constraint suspension*. While the simulation and inference rules are usually kept distinct, for the moment we use the whole collection of them together, in effect a network of constraints that can indicate whether we have a consistent set of assignments to the inputs and outputs. If, for example, we were to try to assign to the network the inputs and *observed* outputs of Fig. 7, it would report a contradiction: there is no way for all the rules to be active (i.e., all the components working as expected) and for those inputs to have produced the observed outputs.

Normally contradictions in constraint networks are handled by retracting one of the values inserted into the network. But here we are sure of the values (the inputs we sent in to the circuit and the outputs we measured); what we are unsure of is the constraints (component behaviors). We

---

7. The guided probe technique, in common use in industry, is based on a set of ideas that is closely related, though not identical. We discuss the differences in Section 13.

therefore take a dual view, and rather than looking for a value we can retract, we look for a *constraint* whose retraction will leave the network in a consistent state. This is the basic idea behind constraint suspension.

To check the global consistency of ADDER-1, for example, we *suspend* (disable) all the rules in ADDER-1, assign the input values to input ports A through E, and assign the *observed* values to output ports F and G. We then allow the whole collection of rules to run to quiescence, determining for us whether there is *some* set of values on the ports of ADDER-1 consistent with the inputs and observed outputs.

If the network does reach a consistent state, we know that the candidate can account for the symptoms. In addition, we can examine the resulting state to see what values the candidate must have at its ports. In the case of ADDER-1, for example, the network indicates that the inputs must be 6 and 6, and the output 10. Thus in the process of determining the global consistency of a candidate, constraint suspension also produces symptom information about the misbehavior.

If the network reports an intractable contradiction, there is no assignment of values to the component that is consistent with all the symptoms, and hence no way for that component to account for the observed malfunction. For example, when we disable the rules of MULT-2 and insert the inputs and observed outputs, an inescapable contradiction results. This demonstrates that MULT-2 cannot account for all the observed values.

Fig. 8 provides a complete description of the candidate generation process in a code-like notation (the procedure has been made easier to follow by ignoring the simulation/inference distinction for the moment and assuming we have a traditional constraint network). Candidate generation occurs in three basic steps: simulate the circuit and collect discrepancies, determine potential candidates using the dependency records; and finally, for each potential candidate determine global consistency and symptom values by using constraint suspension.

```
                    CANDIDATE GENERATION PROCEDURE

STEP 1: COLLECT DISCREPANCIES
        1.1 Insert device inputs into the constraint network inputs
            ; eg. insert 3, 2, 2, 3, and 3 at primary inputs A through E
            ; simulation predicts values at F and G
        1.2 Compare predicted outputs with observed and collect discrepancies
            ; e.g., prediction and observation differ at F.

STEP 2: DETERMINE POTENTIAL CANDIDATES VIA DEPENDENCY RECORDS
        2.1 For each discrepancy found in Step 1:
               follow the dependency chain back from the predicted value
               to find all components that contributed to that prediction
            ; these are all the components "upstream" of the discrepancy
            ; e.g., if we follow the dependency chain back from the 12
            ; at F, we find ADDER-1, MULT-1, and MULT-2
        2.2 Take the intersection of all the sets found by Step 2.1
            ; this yields the components common to all discrepancies (and
            ; hence potentially able to account for all discrepancies)
            ; (in the example above there is only one discrepancy)

STEP 3: DETERMINE CANDIDATE CONSISTENCY VIA CONSTRAINT SUSPENSION
        3.1 For each component found in Step 2.2:
            3.1.1 Turn off (suspend) the constraint modeling its behavior
            3.1.2 Insert observed values at outputs of constraint network
                  ; (inputs were inserted earlier at Step 1.1)
            3.1.3 If the network reaches a consistent state
                  -- the component is a globally consistent candidate
                  -- its symptoms can be found at its ports
                  -- add the candidate and its symptoms to the candidate list
                     ; e.g., ADDER-1 and its values of 6, 6, and 10
                  otherwise
                     the candidate is not globally consistent, ignore it
                     ; e.g., MULT-2
            3.1.4 Retract the values at constraint network outputs
            3.1.5 Turn on the constraint turned off in Step 3.1.1
                  ; (these last two just get ready for the next iteration of 3.1)
```

Figure 8 - Candidate generation via constraint suspension (assuming single point of failure)

As we explore further below, there are a number of important assumptions underlying this reasoning. But constraint suspension provides a mechanism that is both very useful and characteristic of a basic theme underlying this work: the careful management of assumptions. The traditional approach to diagnosis proceeds by assuming that it knows how the component might be failing: it is displaying one of the known misbehaviors found in the set of fault models. We, on the other hand, *proceed by simply suspending all assumptions about how a component might be behaving.* We then allow the symptoms to tell us what the component might be doing. By suspending the constraint in the adder, for example, we are in effect withdrawing all preconceptions about how that component is behaving. We then let the symptoms and the rest of the network tell us whether there is any behavior at all that is consistent with all our observations.

6.2 Advantages of Discrepancy Detection and Constraint Suspension

The combination of discrepancy detection and constraint suspension provides a very useful mechanism with a number of advantages:

† It is 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.

† It reasons from the structure and behavior of the device: the candidate generation process works from the schematic itself to determine which components might be to blame.

† Since it defines failure behaviorally, i.e., as anything that doesn't match the expected behavior, it can deal with a wide range of faults, including any systematic misbehavior. This is more widely applicable than a fixed set of models like stuck-ats.

† As we saw above, the technique yields symptom information about the malfunction: if ADDER-1 is indeed the culprit, then we know a little about how it is misbehaving. As will become clear, this information turns out to be useful in several ways.

† The approach allows natural use of hierarchical descriptions, a marked advantage for dealing with complex structures.

In the example above, for instance, we determined the relevant candidates at the current, fairly high level of description, never having to deal with lower level descriptions (e.g., gate-level devices). We could now continue the process "inside" either candidate, using the next level of description in exactly the same fashion, to determine what subcomponents might be responsible.

† Continuing the process at the next level might indicate that no subcomponent could be responsible, ruling out that candidate.

We might, for example, find that, given how the adder is implemented, there may no subcomponent of it that can logically account for the "6 plus 6 equals 10" symptom that the adder would have to be displaying. Thus the same candidate generation machinery will either provide a set of candidate subcomponents at the next level, or indicate that none can account for the inferred misbehavior, exonerating this candidate.

† This approach keeps knowledge about logical plausibility distinct from knowledge about physical plausibility. This helps simplify construction of the system.

Constraint suspension answers the question of logical plausibility of a candidate: it determines whether there is *any* set of values the component might display that could account for all the symptoms. The technique (by design) knows nothing about whether that set of values is in fact physically plausible.

Our candidate generation machinery would, for example, consider a forked wire to be a plausible candidate if it inferred that the values at its three ports were 1 at the "input" (the point where some device is driving the wire), and 0 and 1 at the two "outputs" (where the wire in turn drives two other devices). Viewed at the black box level, the wire is a three port device that could well display the symptoms noted. To know that this is implausible requires understanding the physics of a specific technology: a wire will display different values at its ends as a result of breaks, but a broken wire in TTL will manifest as a high. Hence the pattern of values given can be ruled out by using knowledge of the particular technology.

The candidate generator thus provides a list of logically plausible components;

further pruning of this list can then be done by invoking a distinct body of technology specific knowledge. Keeping the two distinct simplifies the construction of both.

† The technique extends in straightforward fashion across multiple tests.

Each test of the overall device provides one set of symptoms specifying the misbehavior of a candidate component. The test in Fig. 7, for instance, gives us one I/O pair for ADDER-1 and one for MULT-1; subsequent tests can provide additional I/O pairs. Two kinds of knowledge then relate results across tests.

The non-intermittency assumption indicates that if a component is misbehaving, that misbehavior is at least consistent and reproducible. Hence if a candidate has identical inputs in a later test, it must have an identical output. If, for example, a later test were to indicate that ADDER-1 was a candidate with inputs 6 and 6, and an output of 13, those two test results (and the non-intermittency assumption) exonerate ADDER-1.

The second source of knowledge comes from additional information concerning faults physically plausible within a specific technology. Consider a situation in which Test 1 indicates that a particular wire is a candidate because it is getting a 0 but propagating a 1, while Test 2 indicates the wire is a candidate because it is getting a 1 but propagating a 0. Considered simply as a two port device, this is logically consistent and the candidate generator will report it as such. But knowledge about TTL circuits tells us that there is no physically plausible fault which will cause a wire to start behaving as an inverter, hence the wire can be exonerated.

† This approach makes the <N>-point-of-failure assumption both explicit and easily modified.

In Fig. 8 above the single point of failure assumption appears in two places: at Step 2.2 where we intersected the sets, and at Step 3.1 where we disable the rules for exactly one component when checking candidate consistency. To deal with any specific N > 1, the simplest change is to take the *union* at Step 2.2 and take pairs, triples, etc., at Step 3.1.

While this does not in any sense solve the problem of multiple points of failure, it does demonstrate two important points. First, it shows that, unlike many approaches, our constraint suspension technique is not limited to dealing with a single point of failure. Second, it does provide some aid in grappling with multiple failures, since we can methodically try possible single failures, then pairs, triples, etc.[8]

Unfortunately, as we demonstrate, the power of all of this mechanism is only part of the story.

6.3 Subtleties in Candidate Generation

Consider the slightly revised example shown in Fig. 9. Reasoning as before, we would discover in this case that there is only one hypothesis consistent with the values measured at F and G: the second multiplier is malfunctioning, outputting a 0.

---

8. While *checking* pairs, etc. using constraint suspension (Step 3.1.3) is computationally simple, *generation* of appropriate pairs, triples, etc., (Step 3.1) is still an exponential process. This has not yet proved to be untenable, since both the number of faults considered and the number of components at any given level of description are relatively small. We are nevertheless exploring ways of improving the process. For example, some simple bookkeeping tricks can be used to rule out some of the pairs, triples, etc., very quickly.

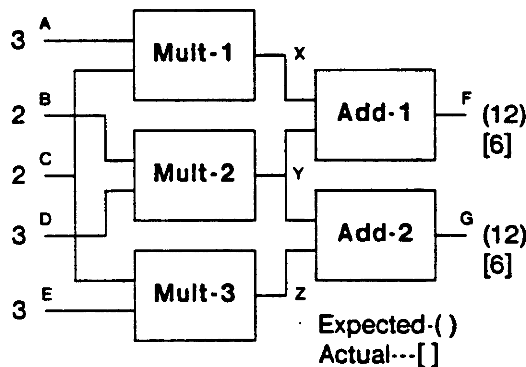

Figure 9 - Troublesome troubleshooting example.

Yet there is another quite reasonable hypothesis: the third multiplier might be bad (or the first).

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 (MULT-3), but the error propagates along an "input" line common to two devices.

The most immediate problem lies in our implicit acceptance of unidirectional ports and the reflection of that acceptance in the basic dependency tracing machinery. We implicitly assumed that wires get information only from output ports --- when checking the inputs to MULT-1, we assumed that the inputs are "primitive". We looked only at terminals A and C, never at the other end of the wire at MULT-3.

Bridges are a second common fault that illustrates another place where we need to be careful: the reasoning style used above can *never* hypothesize a bridging fault, again because of implicit assumptions and their subtle reflection in the reasoning. Bridges can be viewed as wires that don't show up in the design. But we traditionally make 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. The traditional test generation technology suffered from a number of problems: among others, it is a technology for test generation, not diagnosis, and it uses a limited fault model. The use of discrepancy detection and constraint suspension improves on this substantially by providing a diagnostic ability, by defining a fault as anything that produces behavior different from that expected, and by working directly from descriptions of structure and behavior. This seems to be perfectly general, but, as we illustrated, it has to be used with care.

> Put simply, the virtue of the technique is that it reasons from the schematic; the serious flaw in the technique is that it reasons from the schematic *and the schematic might be wrong.*

We believe it is instructive to examine the basic source and nature of this problem.

7. MECHANISM AND KNOWLEDGE

In the example above we encountered some interesting situations because we failed to make explicit a number of important assumptions underlying the reasoning. In the power failure example, we were assuming implicitly that there was only one possible direction of causality at an input port, and thus never examined the other end of wire C. Similarly, tracing back through the circuit from input X of ADDER-1, we looked only at MULT-1, because that was the only apparent connection at that point. We never looked at, say MULT-3, because there was no wire leading there, hence no reason to believe one might affect the other.

Note carefully the character of these assumptions: they concern the *existence of causal pathways, the applicability of a particular model of interaction*. In the power failure example we assumed implicitly that there was no way for MULT-3 to affect MULT-1 through wire C, yet such a path is possible. As we noted above in discussing the possibility of a bridge fault, there seemed to be no path in the schematic that would allow MULT-3 to affect ADDER-1, yet a pathway is in fact possible.

The problem is not in the existence of such assumptions --- they are in fact crucial to the reasoning process. The problem lies instead in the careful and explicit management of them. To see the necessity of having assumptions about causal pathways, consider the nature of the candidate generation task. Given a problem noticed at some point in the device, candidate generation attempts to determine which modules could have caused the problem. To answer the question we must know by what mechanisms and pathways modules can interact. Without *some* notion of how modules can affect one another, we can make no choice, we have no basis for selecting any one module over another.

In this domain the obvious answer is "wires": modules interact because they're explicitly wired together. But that's not the only possibility. As we saw, bridges are one exception; they are "wires" that aren't supposed to be there. But we also might consider thermal interactions, capacitive coupling, transmission line effects, etc.

Generating candidates, then, should not be thought of in terms of tracing wires (or dependency records). Rather, we claim, it should be thought of in terms of *tracing paths of causality*. Wires are only the most obvious pathway. In fact, given the wide variety of faults we want to deal with, we need to consider many different pathways of interaction.

And that leaves us on the horns of a classic dilemma. If we include every interaction path, candidate generation becomes *indiscriminate* --- there will be some (possibly convoluted) pathway by which every module could conceivably be to blame. Yet if we omit any pathway, there will be whole classes of faults we will *never* be able to diagnose.

What can we do? We believe that two steps are important. First, we have to recognize that our inference mechanisms --- in this case dependency detection and constraint suspension --- are not the source of problem-solving power. The power is instead in the knowledge that we supply those techniques, i.e., the pathways of interaction.

And therein lies the second step: there is an important task in enumerating and organizing the pathways of interaction to be considered. How can we do this? We believe that human performance supplies a useful clue.

8. ORGANIZING THE PATHWAYS OF INTERACTION

We appear to be faced with an unavoidable dilemma, caught between the desire to be complete and the need to constrain the possibilities we consider. But people face exactly the same dilemma and seem to handle it. What do they do?

The answer seems be an instantiation of Occam's razor: an experienced engineer knows that some things are more likely to go wrong than others. He will, as a result, attempt to generate solutions that employ simpler and more likely hypotheses first, falling back on more elaborate possibilities only in face of an intractable contradiction (i.e., given the current set of assumptions, there is no way to account for the observed misbehavior). There are three important points here.

The engineer has a notion of "the kinds of things that can go wrong".

There is an ordering criterion that indicates which category of hypotheses to entertain first.

The categories are ordered but none is permanently excluded.

To capture this same sort of behavior in our program we need to (i) make precise the notion of "what can go wrong", and (ii) determine what constitutes a "simple" explanation. We consider both of these briefly here, as background for the example that follows, then address the issue in detail in Section 10.

To address the first of these, we need some methodical way to define and generate the possible kinds of failures. This is accomplished by enumerating the assumptions built into our "module and information path" representation and then characterizing the variety of failure that results from violating each assumption. We refer to the resulting list as the *categories of failure*.

One such category is illustrated by the problem presented in Fig. 9. The implicit assumption there was that information flows in only one direction at an input (or output) port. If this assumption is violated, we get a category of failure we term an "unexpected direction" failure. The other categories generated in this way are described in Section 10.

Given such a notion of "what can go wrong", we now need an appropriate metric for ordering the list. This is currently accomplished by relying on the experience of expert troubleshooters, who tell us which categories of failure are encountered more frequently than others. Stuck-ats are more likely than assembly errors, for example. While the ordering criterion may eventually need be more elaborate, its precise content is less an issue here than its character: it is a summary of empirical experience that helps us to order the kinds of hypotheses we consider. For our current domain, this approach produces the following list:

localized failure of function (e.g., stuck-at on a wire, failure of a RAM cell)
bridges
unexpected direction (e.g., the power failure problem)
multiple point of failure
assembly error
design error

We start by attempting to generate candidates in the localized failure category, assuming that the structure is as shown in the schematic, that there was only a single point of error, that information flowed only in the predicted directions, etc. Only if this leads to a contradiction are we willing to surrender an assumption (e.g., that the schematic was correct) and entertain the notion that a bridge might be at fault. If this too leads us down a blind alley then we would surrender additional assumptions and consider ever more elaborate hypotheses, eventually entertaining the possibility of multiple errors, an assembly error (every individual component works but they have been wired up incorrectly) and even design errors (the implementation is correct but cannot produce the desired behavior).

This mimics what we believe a good engineer will do: make all the assumptions necessary to simplify a problem and make it tractable, but be prepared to discover that some of those simplifications were incorrect. In that case, surrender some of those assumptions and be willing to consider additional kinds of failure.

In terms of the dilemma noted above, the categories of failure serve as a set of

filters. They restrict the paths of interaction we are willing to consider, thereby preventing candidate generation from becoming indiscriminate. In using the localized failure of function category, for example, we are assuming for the moment that the structure is as shown in the schematic, hence there are no additional paths of interaction (and thus no bridges). But these are filters that we have carefully ordered and consciously put in place. If we cannot account for the observed symptoms with the current set of filters in place, we remove one, leaving us with a set that is less restrictive, allowing us to consider additional interaction paths and hence more elaborate hypotheses.

There are of course no guarantees that this will lead us to the correct category of failure without any false steps. It is possible that all the evidence at a given point is consistent with, say, a localized failure of function in a single component, yet replacing that component may make it clear that the fault is elsewhere. As always with Occam's razor, our only assurance is that we are generating a hypothesis that is by some measure the simplest and most likely, and that's the best we can do. We may subsequently discover that the problem is in fact more complex. By making the ordering criteria explicit and accessible, we have at least provided a place for embedding knowledge that can make the choice of hypothesis category as informed as possible at each step.

9. EXAMPLE: A BRIDGE FAULT

As we have noted, traditional automated reasoning about circuits works from a predefined list of fault models and uses the mathematical style of analysis exemplified by boolean algebra or the D algorithm. As a result, it is strongly oriented toward faults that can be modeled as a permanent binary value. One problem with this is its inability to provide useful results concerning bridge faults.

In this section we show how our system works when faced with a bridge fault, illustrating a number of the ideas described above. While the example has been simplified for presentation, there is still unfortunately a fair amount of detail necessary. A summary of the basic steps will help make clear how the problem is solved.

> The device is a 6-bit carry-chain adder (Fig. 10). The problem begins when we notice that the attempt to add 21 and 19 produces an incorrect result.
>
> The candidate generation process outlined above generates a set S1 of three candidates, any of whose malfunction can explain this result.
>
> A new set of inputs (1 and 19) is chosen in an attempt to discriminate among the three possibilities. The adder's output is incorrect for this set of inputs also. The candidate generator indicates that there are two candidates capable of explaining this new result.
>
> Neither of these two candidates are found in S1. Thus we reach a contradiction: no one component is capable of explaining the data from both sets of inputs.
>
> Put slightly differently, we have a contradiction *under the current set of assumptions and interaction models*. We therefore have to surrender one of our assumptions and use a different interaction model.
>
> The next model --- bridge faults --- surrenders the assumption that the structure is as shown in the schematic and considers one class of modifications to the structure: the addition of one wire between physically adjacent pins.
>
> The combination of functional information (the expected pattern of values produced by the fault) and physical adjacency provides a strong constraint on the set of connections which might be plausible bridges.
>
> The first application of this idea produces two hypotheses that are functionally plausible, but both are ruled out on physical grounds.
>
> Dropping down a level of detail in our description reveals additional bridge candidates, two of which prove to be both physically and functionally plausible. One of these proves to be the actual error.

A key point is the utility of ordering the paths of interaction to be considered. Starting with a very restricted category of failure, we discover that it leads us to a contradiction. We surrender an assumption, consider an additional category and hence an additional pathway of interaction: bridge faults. We show how knowledge of both structural and functional organization allows us to generate a select few bridge fault hypotheses, eventually discovering the underlying fault.

9.1 The Example

Consider the six bit adder shown in Fig. 10 and imagine that the attempt to add 21 and 19 produces 36 rather than the expected value of 40. Invoking the candidate generation process described above, we would find that there are three devices (SLICE-1, A2 and SLICE-2), any one of whose malfunction can explain the misbehavior.[9]

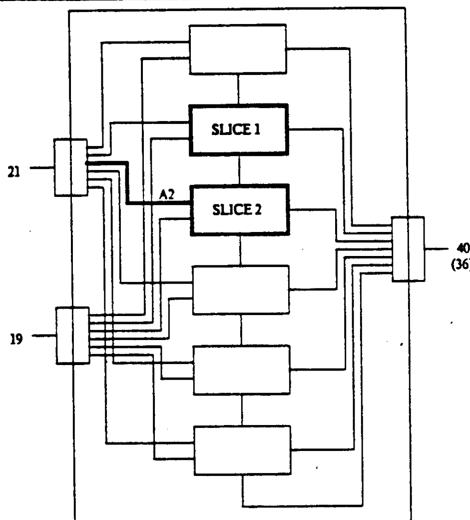

Figure 10 - Six bit adder constructed from single bit slices. Heavy lines indicate components implicated as possibly faulty.

A good strategy when faced with several candidates is to devise a test that can cut the space of possibilities in half. In this case changing the first input (21) to 1 will be informative: if the output of SLICE-2 does not change (to a 0) when we add 1 and 19, then the error must be in either A2 or SLICE-2.[10]

As it turns out, the result of adding 1 and 19 is 4 rather than 20. Since the output of SLICE-2 has not changed, it appears that the error must be in either A2 or SLICE-2.

But if we invoke the candidate generator, we discover an oddity: the only way to account for the behavior in which adding 1 and 19 produces a 4 is if one of the two candidates highlighted in Fig. 11 (B4 and SLICE-4) is at fault.

9. The example has been simplified slightly for presentation.

10. This and subsequent test generation is currently done by hand. Work on automating test generation is in progress [27].

The reasoning behind this test relies on the single fault assumption: if the malfunctioning component really were SLICE-1, both A2 and SLICE-2 would be fault-free. Hence the output of SLICE-2 would have to change when we changed one of its inputs. (Notice, however, if the output actually does change, we don't have any clear indication about the error location: SLICE-2, for example, might still be faulty.)

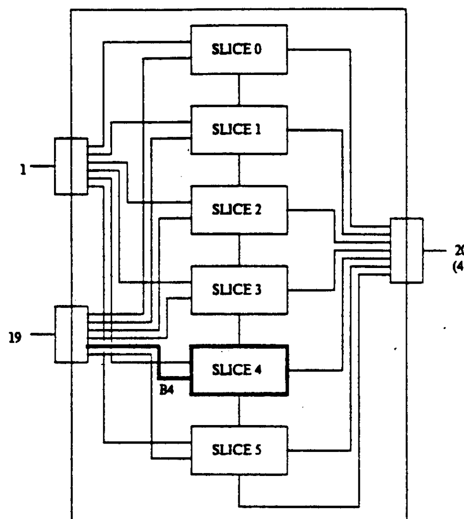

Figure 11 - Components indicated as possibly faulty by the second test.

Therein lies our contradiction. The only candidates that account for the behavior of the first test are those in Fig. 10, the only candidates that account for the second test are those in Fig. 11. There is no overlap, so there is no single candidate that accounts for all the observed behavior.

Our current category --- the localized failure of function --- has thus led us to a contradiction.[11] We therefore surrender it and consider the next, less restrictive category, one that allows us to consider an *additional kind of interaction path* --- bridging faults. The problem now is to see if there is some way to unify the test results, some way to generate a single bridge fault candidate that accounts for all the observations.

Much of the difficulty in dealing with bridging faults arises because they violate the rather basic assumption that the structure of the device is in fact as shown in the schematic. But admitting that the structure may not be as pictured says only that we know what the structure *isn't*. Saying that we may have a bridge fault narrows it to a particular class of modifications to consider, but the real problem here remains one of *making a few plausible conjectures about modifications to the structure*. Between which two points can we insert a wire and produce the behavior observed?

To understand how we answer that question, consider what we have and what we need. We have test results, i.e., observations of *behavior*, and we want conjectures about modification to *structure*. The link from behavior to structure is provided by knowledge of electronics: in TTL, a bridge fault acts like an and-gate, with ground dominating.[12]

From this fact we can derive a simple pattern of behavior indicative of bridges. Consider the simple example of Fig. 12 and assume that we ran two tests. Test 1 produced one candidate, module A, which should have produced a 1 but yielded a 0 (the zero is underlined to show that it is an incorrect output). Module B was working correctly and produced a 0 as expected. In Test 2 this situation is exactly reversed, A was performing as expected and B failed.

The pattern displayed in these two tests makes it plausible that there is a bridge

11. Note that dropping down another level of detail in the functional description cannot help resolve the contradiction, because our functional description is a tree rather than a graph: in our work to date, at least, no component is used in more than one way. (If the functional description were in fact a graph, we could easily continue down it to see if the two candidate sets did indeed have a subcomponent in common).

12. This is an oversimplification, but accurate enough to be useful. In any case, the point here is how the information is used; a more complex model could be substituted and carried through the rest of the problem. Note also that for notational convenience, we assume in the rest of the description that ground is equivalent to a 0.

linking the outputs of A and B: in the first test the output of A was dragged low by B, in the second test the output of B was dragged low by A.

Figure 12 - Pattern of values indicative of a bridge. Heavy lines indicate candidates.

We have thus turned the insight from electronics into a pattern of values on the candidates. It is plausible to hypothesize a bridge fault between two modules A and B from two different tests if: in test 1, A produced an erroneous 0 and B produced a valid 0, and in test 2, A produced a valid 0 while B produced an erroneous 0. Note that this can resolve the contradiction of non-overlapping candidate sets: it hypothesizes one fault that involves a member of each set and accounts for all the test data.

Thus, if we want to account for all of the test data in the original problem with a single bridge fault, we need a bridge that links one of the candidates from the first test (SLICE-1, A2, SLICE-2) with one of the candidates from the second test (B4, SLICE-4), and that mimics the pattern shown in Fig. 12.

Fig. 13 shows the candidate generation results from both tests in somewhat more detail. As noted earlier, the candidate generation procedure can indicate for each candidate the values that would have to exist at its ports for that candidate to be the broken one. For example, for SLICE-1 to be at fault in test 1, it would have to have the three inputs shown, with its sum output a zero (as expected) and its carry output also a zero (the manifestation of the error, underlined).

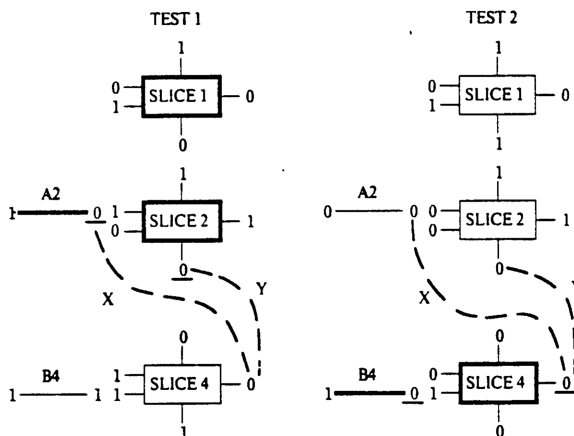

Figure 13 - Candidates and values at their ports.

In Fig. 13 there are two pairs of devices that match the desired pattern, yielding two functionally plausible bridge hypotheses:

Dotted line X, bridging wire A2 to the sum output of SLICE-4;

Dotted line Y, bridging the carry output of SLICE-2 to the sum output of SLICE-4.

But the faults have to be physically plausible as well. For the sake of simplicity, we assume that bridge faults result only from solder splashes at the pins of chips.[13] To check physical plausibility, we switch to our physical representation, Fig. 14. Wire A2 is

connected to chip E1 at pin 4 and chip E3 at pin 4; the sum output of SLICE-4 emerges at chip E2, pin 13. Since they are not adjacent, the first hypothesis is not physically reasonable. Similar reasoning rules out Y, the hypothesized bridge between the carry-out of SLICE-2 and the sum output of SLICE-4.

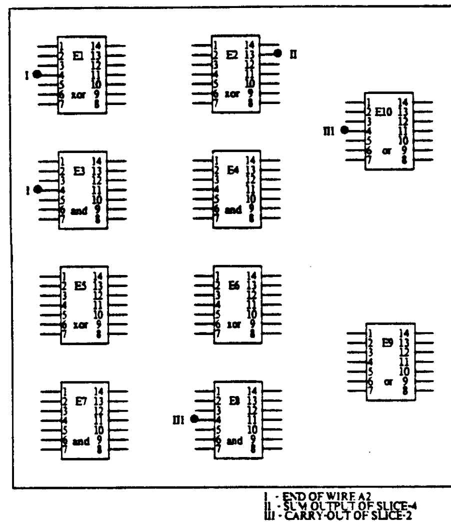

Figure 14 - Physical layout of the board with first bridge hypotheses indicated. (Slices 0, 2, and 4 are in the upper 5 chips, slices 1, 3, and 5 are in the lower 5.)

So far we have considered only the top level of functional organization. We can run the candidate generator at the next lower level of detail in each of the non-primitive components in Fig. 13. (Dropping down a level of detail proves useful here because additional substructure becomes visible, effectively revealing new places that might be bridged.)

We obtain the components and values shown in Fig. 15. Checking here for the desired pattern, we find that either of the two wires labeled A2 and S2 could be bridged to either of the two wires labeled S4 and C4, generating four functionally plausible bridge faults.

13. Again this is correct but oversimplified (e.g., backplane pins can be bent or bridged), but as above we can introduce a more complex model if necessary.

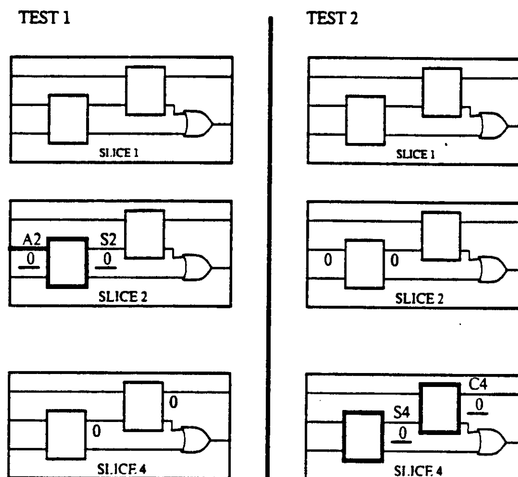

Figure 15 - Candidates at the next level of functional description. Each single bit adder is built from two "half-adders" and an OR gate. (To simplify the figure, only the relevant values are shown.)

Once again we check physical plausibility by examining the actual locations of A2, S2, S4, and C4 (Fig. 16). [14] As illustrated there, two of the possibilities are physically plausible as well: A2-S4 on chip E1 and S2-S4 on chip E2.

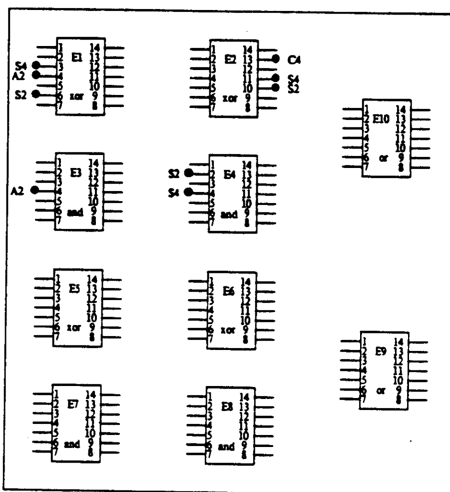

Figure 16 - Second set of bridge hypotheses located on physical layout.

Switching back to our functional organization once more, Fig. 17, we see that the two possibilities correspond to (X) an output-to-input bridge between the xor gates in the rear half-adders of SLICE-2 and SLICE-4, and (Y) a bridge between two inputs of the xor in the forward half-adders of slices 2 and 4.

14. Note that the erroneous 0 on wire S2 can be in any of three physical locations, because S2 fans out (inside the module it enters on its right).

10. CATEGORIES OF FAILURE

Since the categories of failure and paths of interaction play a significant role in our approach, two obvious questions concern their origin and ordering. Where do we get them, and how can we determine an appropriate ordering?

We want a methodical way to generate the categories so that we have some reason to believe that the result is systematic and hence reasonably complete. We need to define the criterion for simplicity so that we know how to order the categories to produce a sequence of successively more elaborate hypotheses.

10.1 Origins

The list in Section 8 of possibilities to consider is of course specific to our current domain. We believe that our overall framework does, however, offer a useful set of questions that we can ask about any domain to generate an analogous listing. Appropriately enough, many of the questions derive from examining carefully our simple "module and information path" representation, asking what assumptions the representation makes about the world, and then asking what the consequences are of violating those assumptions.

The simplest assumption (indicated schematically in Fig. 18a) lies in believing that a module can be said to have a particular behavior. To say this is violated means, as we saw earlier, that the module isn't behaving in accordance with the assigned behavior, i.e., it is broken. This is the simple "localized failure of function" category that heads the list in Section 8. Simple examples in our current domain include the traditional stuck-at (a particular kind of failure of a wire module), as well as failures of primitive gates (e.g., a NOR-gate acting as an inverter), or even the use of an incorrect part during assembly (e.g., a NOR-gate chip instead of a NAND-chip).

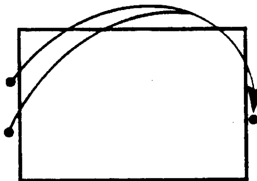

Figure 18a - Assigning behavior to a module.

Our representation also assumes that modules have ports, each with a specified direction (Fig. 18b). Yet as we saw above, this too can be violated. We term it an "unexpected direction" error, with the power failure example as one instance.

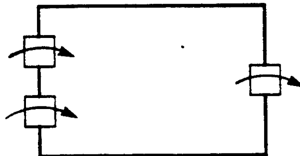

Figure 18b - Assuming ports have specified direction.

Treating a module as a black box means committing only to its behavior; when we drop down a level in the description, we are assuming that it has a specific substructure. But that may not be true; we refer to this as a "structure error" (Fig. 18c). How could the substructure be different from what we expect? In our current domain there are a

choosing a particular variety of adjacency, contiguity of pins on chips. This produced a significant reduction in the potential search space, but still left us with too many choices to produce an effective hypothesis generator: trying each pair of pins would be too unwieldy.

We then found a way to reduce the search using the functional representation. We used knowledge about electronics to derive a link from behavior to structure, producing a pattern characteristic of bridges. This reduced the search space considerably, since the pattern had to include one candidate from each of the two sets (each of which is itself typically small), and since the pattern of "alternating 0's" is relatively rare.

The result was sufficiently constrained to be an effective generator of bridge hypotheses. We were then able to use the physical representation as a filter on the hypotheses generated.

In general then, our system produces a focused development of its solution by relying on several keys:

> The layered set of interaction models produced by methodical enumeration and relaxation of underlying assumptions constrained the categories of errors we were willing to consider, yet allowed us to consider more elaborate hypotheses when simpler ones failed.
>
> The availability of multiple representations allowed us to take advantage of constraints associated with each representation.
>
> We were able to use one representation as the basis for a constrained generator of hypotheses and use the other to filter the hypotheses generated.

In Section 11 we speculate on ways of generalizing the set of ideas used here. We consider the character of the representations used and ask what made them effective generators and filters. Our goal there is to produce a set of principles that will function as guidelines in selecting representations, making it possible to carry over this approach to other problems in other domains.

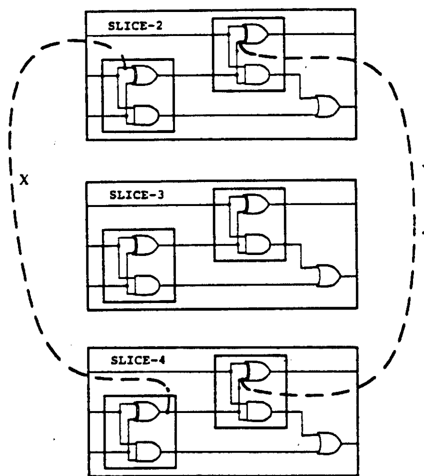

Figure 17 · Functional representation with bridge fault hypotheses illustrated.

It is easy to find a test that distinguishes between these two possibilities[15] : adding 0 and 4 means that the inputs of SLICE-2 will be 1 and 0, with a carry-in of 0, while the inputs of SLICE-4 will both be 0, with a carry-in of 0. This set of values will show the effects of bridge Y, if it in fact exists: the sum output of SLICE-2 will be 0 if it does exist and a 1 otherwise. When we perform this test the result is 1, hence bridge Y is not in fact the problem.

Bridge X becomes the likely answer, but we should still test for it directly. Adding 4 and 0 (i.e., just switching the order of the inputs), is informative: if bridge X exists the result will be 0 and 1 otherwise. In this case the result is 0, hence the bridge labeled X is in fact the problem.[16]

9.2 The Example: Summary and Comments

A fundamental point illustrated by the example is the utility of a layered set of models as a device for making explicit our simplifying assumptions and for dealing with complexity in candidate generation. Our original model, localized failure of function, incorporated the largest set of simplifying assumptions and was the most restrictive. It worked initially, but we eventually found ourselves unable to account for all the observations. At that point we surrendered one of our assumptions, adopted a less restrictive model, and considered an additional path of interaction. This allowed us to generate several bridge fault hypotheses, one of which eventually proved to be correct. The notion of layers of interaction models thus provided an important overall framework guiding the problem solving.

A second source of problem solving power came from using multiple representations. This was particularly important in constraining the generation of bridge candidates. Faced with the original, apparently insoluble problem, we admitted the possibility of a bridge. But this left us with the difficult task of deciding where the bridge might be. The problem is quite similar to adding a new line to complete a proof in geometry and the difficulty is analogous: new constructions are difficult in general because they are relatively unconstrained [22].

The system's search for likely candidates turned out to be focused because we were able to derive useful constraints from each of our multiple representations. Consider the physical representation, where the constraint is physical adjacency. We started by

15. As above, tests are currently generated by hand.

16. Had both been ruled out by direct test, then we would once again have had a contradiction on our hands and would have had to drop back to consider a still more elaborate model with additional paths of interaction.

wide range of answers, i.e., a wide range of paths of interaction: bridges, thermal or transmission line effects, as well as out of date schematics or a wiring error during assembly. Each of these has the effect of producing a substructure different from what we expected.

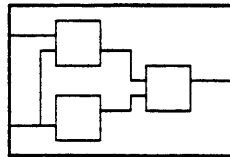

Figure 18c - Assuming substructure is as specified.

Finally, our representation assumes that the overall behavior of a module should be matched by the aggregate behavior of its substructure (Fig. 18d). Violating this assumption means there is a design error: all components work as specified, but they cannot produce the desired behavior.

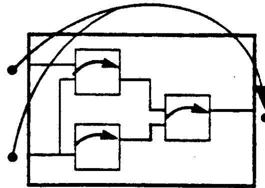

Figure 18d - Assuming overall behavior matches aggregate of substructure.

Thus, examining carefully the character of the assumptions built into our representation yields a set categories of failure. These in turn provide a number of questions we can ask about the domain, questions whose answers are the pathways of interaction to be considered. While we have answered these questions here for our digital circuits, we believe the questions are more broadly applicable, and may prove to be a useful way to think about other domains where the "module and information path" view is appropriate.

One additional category derives from examining our candidate generator. As noted in Section 6.2, the single point of failure assumption is embodied in the fact that we disabled the rules for exactly one component when checking candidate consistency. This too may be an incorrect assumption, yielding yet another category, multiple faults.

10.2 Ordering

The example in Section 9 used a very simple approach to ordering the categories: a fixed order based on frequency of occurrence as reported by experienced troubleshooters. While the program currently uses this approach, it is easy to imagine more sophisticated ordering schemes that still fit naturally into this framework. The remainder of this section speculates about a number of such possibilities.

More elaborate static ordering criteria might take account of the age of the machine or the age of the design. We might want to indicate that assembly errors are more common in machines received from the production line recently, and design errors more likely if the device is a new model or has undergone substantial redesign.

Note that, were we willing to model enormously more of the world, we might be able to *infer* that design errors are less common in well-established designs, but this is

beyond the scope of our efforts and the investment would be very large for a relatively modest return. As it stands, we are in this case willing to rely on some simple experiential observations for guidance, without bothering to model the causality in any detail.

The character of the reported misbehavior can be revealing as well. Small changes in external behavior are often suggestive of the local failure of function category, while substantial variations in behavior can be suggestive of assembly and wiring errors. We might thus borrow from medicine the notion of a "presenting complaint" and use information about the character of the misbehavior to reorder the categories to be considered.

An obvious further extension would make the ordering dynamic. We might start out as above, but reorder the categories in response to information gained as inference and testing proceeds.

Whatever the ordering criteria chosen, the overall point is that having an ordered set of fault categories provides a number of advantages. First, it offers a means of expressing and managing simplifying assumptions. Second, it provides a simple mechanism for dealing with complexity, by limiting the set of interaction paths to consider. Finally, it supplies a relatively natural site for embedding and using the empirical knowledge of an experienced engineer that might be difficult to represent at the level of first principles.

11. THE ADJACENCY PRINCIPLE

The bridge fault example raises two interesting questions:

Why are bridge faults so difficult?

Why does the physical representation prove to be so useful?

To see the answer, we start with the trivial observation that all faults are the result of some difference between the device as it is and as it should be. With bridge faults the difference is the addition of a wire between two physically adjacent points.

Now recall the nature of our task: we are typically presented with a device that misbehaves, not one with obvious structural damage. Hence we reason from behavior, i.e., from the functional representation. And the important point is that for a bridge fault, the difference in question --- the addition of a single wire --- is not small and local in that representation. As the comparison of Figs. 16 and 17 makes clear, the new wire connects two points that are adjacent in the physical representation but widely separated in the functional representation.

The difference is also not as simple in that representation: if we include in our functional diagram the AND gate implicitly produced by the bridge (Fig. 19), we see that a single added wire in the physical representation maps into an AND gate and a fanout in the functional representation.

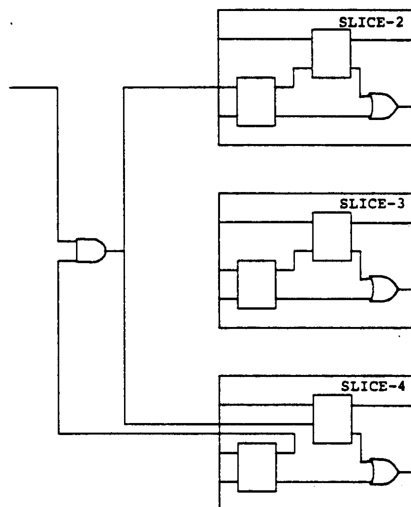

Figure 19 - Full functional representation of bridge fault X.

This view helps to explain why bridge faults produce behavior that is difficult to understand. Bridge faults are modifications that are simple and local in the physical description, but our reasoning is done using the functional description. Hence the dilemma:

The desire to reason from behavior requires us to use a representation that does not necessarily provide a compact description of the fault.

This non-locality and complexity should not be surprising, since devices physically adjacent are not necessarily functionally related. Hence there is no guarantee that a change that is compact in one will produce a change that is compact in the other. More generally, *changes compact in one representation are not necessarily compact in another.*

We can turn this around to put it to work for us:

Part of the art of choosing the right representation(s) for diagnostic reasoning is finding one in which the suspected change is compact.

This explains the utility of the physical representation: it's the "right" one because it's the one in which the change is compact because the affected components are adjacent.

Going one step further, we might ask why adjacency is the relevant organizing

principle. We believe the answer follows from two facts: (a) devices interact through physical processes (voltage on a wire, thermal radiation, etc.) and (b) physical processes occur locally, or more generally, causality proceeds devices that are in some sense adjacent: there is no action at a distance. To make this useful, we turn it around:

The paths of interaction are one way to define adjacency. That is, each kind of interaction path can define a representation.

Bridge faults arise from *physical adjacency* and hence are local in the physical representation. The notion of *thermal adjacency* and a corresponding representation would be useful in dealing with faults resulting from heat conduction or radiation, *electromagnetic adjacency* would help with faults dealing with transmission line effects, etc.

Each of these produces a different representation, different in its definition of adjacency. And each will be useful for understanding and reasoning about a category of failure.

The paths of interaction are in effect a set of "representation generators". They can tell us what kinds of views to take of the device; how to represent it in order to make use of the fundamental belief that physical action proceeds between components that are adjacent in some way.

Finally, we can push this one step further, asking why we expect that there will in fact be some representation, some definition of adjacency, that "makes sense" of the problem at hand. Why should there be some representation in which the fault appears to be a compact perturbation? The belief appears to be based on what we might call the "single initial cause heuristic":

It is often the case that the malfunction of a previously working device results from a single cause, rather than a number of simultaneous, independent events.

When there is indeed a single cause, there will be some representation in which a small and local change accounts for the difference between the good and faulty device. To the extent that a malfunction results from multiple, *unrelated* events (e.g., multiple failures in an old machine; multiple problems inserted deliberately), it becomes far less likely that we can find a single representation that offers a unifying perspective for every fault at once.[17]

One additional illustration of the utility of multiple representations with multiple definitions of adjacency comes from considering devices with shared components. In some designs, one physical component (e.g., an AND-gate) may be used for two different purposes. We can model this by having two different modules in the functional representation that point to the same module in the physical representation (using the cross-links shown in Fig. 4). If the physical gate is bad, then the candidate generator may find that it needs two points of failure to account for the symptoms. But what looks like two failures in the functional representation can be resolved to a single point of failure by shifting to the physical representation.

Similarly, if a chip containing 4 AND gates is completely bad, our candidate generator will find four apparently distinct candidates. Once again, shifting to the physical representation gives us a view in which the problem can be understood as a single failure.

In both cases shifting to a different definition of adjacency helps resolve the issue and provides alternate definitions of the notion of a single point of failure. In the first case two modules distant in functional representation are adjacent (in fact coincident) in the physical representation; in the second case four different functional modules become adjacent (same package) in the physical representation.

As this illustrates, the basic concept of a "single point of failure" is not in fact well defined until we specify the representation. We should really speak of a single point with respect to a particular representation, since, as we have seen, what appears as multiple distinct points in one representation may be a single point in a different

17. This work has not yet explored the problem of multiple, related failures, where one fault causes a cascade of other failures.

representation.

We still have substantial additional work to do on these topics, but we seem at least to be asking the right questions. The concept of pathways of interaction and the corresponding definitions of adjacency appear to make sense for a wide range of hardware faults.

The ideas also appear to work in other domains. When debugging software, for example, the pathways of interaction differ (e.g., procedure call, mutation of data structures), but the resulting perspectives appear to make sense and there are some interesting analogies. Unintended side effects in software, for example, are in some ways like bridge faults. An unexpected direction of information flow can result during a procedure call, by assigning a value to a parameter that turns out to be called by name rather than by value.

More generally still, there appears to be substantial breadth to this whole perspective of viewing different representations as embodying different definitions of adjacency. Consider software once again. Our typical view of software is functional, in flowcharts or dataflow diagrams. Yet there are faults where the notion of "physical adjacency" is crucial in understanding the bug. Consider for example an out-of-bounds array addressing error. Full understanding of the bug may require knowing how the software is mapped physically into memory, so that we can determine which cell was actually referenced.

A similar phenomenon is apparent in speech: the classic "recognize speech" vs. "wreck a nice beach" confusion is entertaining in part because the two are so "far apart" in meaning (distant in any semantic representation) yet adjacent in their phonetic representation. The idea even supports "troubleshooting" in this domain. In reading an abstract recently, I encountered a number of phrases that made no sense in an otherwise coherent text. Then, considering the origin of the document, it became plausible that the text might have been dictated over a noisy (transatlantic) phone line. This suggested that a phonetic representation might be the "correct" one for debugging, i.e., the one in which the errors could be seen as small and local changes. It was then easy to debug phrases like "decorative representations" ("declarative representations") and "rule space systems" ("rule based systems").

Two widely accepted bits of wisdom about AI suggest that having the "right" representation is an important key in problem solving and that multiple representations often contribute substantial power. We believe that the notion of adjacency as a way of defining representations is useful in understanding more about both of these ideas. In the effort to define what is meant by the "right" representation, for example, a number of general guidelines have emerged, at the level of suggestions that appropriate representations "make the important things explicit", and "expose the natural constraints" (e.g., [34]). The notion of adjacency offers one additional level of detail to these catchphrases. It suggests that natural constraints arise because things are adjacent "in some sense", and that "in some sense" can have multiple different instantiations. We can generate a variety of representations by seeking out different definitions of adjacency. Finally, this in turn may help explain the utility of multiple representations: part of the power in using them arises because they provide us with multiple different definitions of adjacency.

12. EVALUATION AND FUTURE WORK

Since this work is still in its formative stages, there are still a number of interesting questions about its limits and appropriate directions for future work.

12.1 Advantages of Reasoning From Structure and Behavior

There are several advantages in basing reasoning on knowledge of structure and behavior, rather than previous approaches like the empirical associations used in a number of earlier programs (e.g., MYCIN [28.9], INTERNIST [24], or PROSPECTOR [17]). Reasoning from first principles provides a strong degree of machine independence, makes the system easier to construct and maintain, facilitates defining the program's scope of competence, and makes it capable of dealing with bugs that display novel symptoms.

Reasoning from first principles offers a significant degree of machine independence. The tools and techniques described above work directly from the descriptions of structure and behavior and make explicit their assumptions about those descriptions. This allows us to cover a wide range of devices built from digital logic components. Even more fundamentally, concepts like modules linked by causal pathways, expected behavior, etc., are widely useful, extending perhaps to include handheld calculators, digital watches, automobiles and software.

Rules, on the other hand, are typically strongly machine-specific. There is distressingly little carry-over from one machine to the next when we capture troubleshooting knowledge at the level of symptom/disease associations. Even simple changes to a single machine (e.g., design upgrades) can mean substantial changes in the rules. Working on a new machine means even more difficulty. Each new system would require a new knowledge base with all of the difficulties that entails.

A system based on reasoning from first principles is easier to construct because there is a way of systematically enumerating the required knowledge: the structure and behavior of the device. A system based on empirical associations is more difficult to construct because the character of the knowledge makes it necessary to extract the rules on a case by case basis. To the extent that the knowledge is a distillation of the expert's experience, the best we can do is to assemble a representative collection of cases and ask an expert for the rules dealing with each case. No more systematic method of collecting rules is available and the process often continues for an extended period of time. This time lag is a particular problem in dealing with electronic hardware: the time necessary to accumulate the relevant experience is beginning to be longer than the design cycle for the next model of the machine.

A system based on reasoning from first principles is also easier to maintain, since modifications to the machine design are relatively easy to accommodate. We can update the structure and behavior specifications for each modified component, rather than having to determine how each change should modify the overall behavior and the troubleshooting strategy.

The ability to enumerate the knowledge systematically also aids in defining the program's scope and competence. We know what parts of the machine have been modeled and to what level of detail.

When building a system from empirical associations, it is more difficult to define precisely what such a program "knows". A precise answer to the question is often possible only for a problem that can be formalized, so almost every AI system suffers from this to some extent. But the more systematic our mechanism for enumerating the required knowledge, the more precise we can be about defining the program's competence. Deriving the knowledge strictly from a collection of case studies is one of the less systematic mechanisms, yet it is in such cases that rule-based systems are appropriate. As a result, often the best we can say about the system is that it has a set of rules dealing with one or another class of problems, with only a very informal measure of how thoroughly that class has been covered.

Finally, reasoning from first principles offers the possibility of dealing with novel

faults. As we have seen, our system does not depend for its performance on a catalog of observed error manifestations. Instead it takes the view that any discrepancy between observed and expected behavior is a bug, and it uses knowledge about the device structure and behavior to determine the possible sources of the bug. As a result, it is able to reason about bugs that are novel in the sense that they are not part of the "training set" and are manifested by symptoms not seen previously.

Since rules are a distillation of an expert's experience, a program built from them will be reasonably sure of handling only cases quite similar to those the human expert has already seen, solved, and communicated to the program. We have little reason to believe that the program will handle a bug whose outward manifestation is unfamiliar, even if the root cause is within the claimed scope of the system.

At times there may not be a choice: in domains where knowledge truly is anecdotal and experiential, a rule-based encoding may be appropriate. Where knowledge in the domain does admit of some more basic model, however, we find the reasons above present a strong argument for capturing that level of understanding in the program.

12.2 Describing Structure and Behavior

12.2.1 Previous representations

There is a long history of attempts to represent structure and behavior in general and computer hardware in particular. Our approach has a number of features in common with that work. The primary points of overlap are the black box view of modules and the use of hierarchical descriptions (both of which are found in [16] and [33], for example). Major points of difference include (i) our distinction between, and explicit representation of functional, physical and behavioral information, (ii) emphasis on the creation of a domain-specific language, and (iii) the ability to make multiple uses of our representations.

Many previous languages fail to include information about physical organization at all. This seems to have resulted from their origin in work on machine design, where physical packaging is sometimes considered only after functional design has been accomplished.

In some other description languages there is a significant intermixing of structure and behavior information, arising apparently from the use of traditional programming languages as foundations. There are many advantages to implementing a new language as a variant on some existing language (like ALGOL), including the availability of a compiler and other language development tools. But this made it all too easy to carry over a set of habits that may be inappropriate. The temptation exists, for example, to use datatype declarations as a mechanism for expressing the existence of functional modules. Hence we see such things as

```
STRUCTURE:

        DECLARE
        MEMORY[0:1023]<0:15>
        ACCUMULATOR<0:16>
        PC<0:11>
        IR<0:15>;

BEHAVIOR:

        IR := MEMORY[PC]
        IF IR<8:11> =   13        ; add instruction
        ...
        etc.
```

There is a significant intermixing of structure and behavior here. The declarations are supposed to contain the structure information, but where is the indication that the

instruction register (IR) is connected to the memory? It is indicated indirectly in the behavior section of the code, where we discover that the IR is capable of being loaded from memory. We prefer that such information be made explicit. Among other things, this permits more systematic fault insertion. In the current example, we would be able to simulate the effects of faulty behavior in the bus that links the memory and instruction register.

A second major difference in our approach lies in our creation of a domain-specific language, one that incorporates the concepts and vocabulary of computer architecture. As noted, several hardware description languages have used the the black box view and the module/port approach as a starting point. Simple interconnections of devices are easily accommodated in this scheme, but the attempt to describe even small real circuits soon presents problems. One common insight is that many circuits contain a substantial amount of regularity. The pattern of interconnections in memory or between slices of an ALU, for example, is often easily expressed as some form of iteration. One common response (see, e.g., [21]) is to adopt more of the traditional programming language constructs, like iterative loops.

But this presents problems. A bitslice CPU, for example, has a form of regularity different from that displayed by an array of memory chips. Using a **for** loop to express both of them ignores, or at best makes obscure, the important difference. The result is often difficult to interpret as well. It can take a considerable amount of study to determine that a tightly coded iterative loop in fact expresses a well-known organization.

We are pursuing an alternate approach, described in Section 3, of assembling a vocabulary of terms common to architects. Each such term labels a particular kind of organization and has associated with it information on how to wire up that configuration. The result is in effect a new, high-level language, with all of the standard benefits of such. It allows us to make explicit the nature of the organization in the circuit, as well as providing a compact, efficient, and easily understood language.

A third major difference in our approach is our ability to make use of descriptions in several different ways. The same description is used (i) as a basis for the troubleshooting module, (ii) as a database of facts about connectivity, part identity, etc., (iii) as a body of code that can be run to simulate the device, and (iv) as the basis for a display program for observing the device. This is possible because we have avoided the temptation to write code oriented toward a single purpose like simulation, and have instead produced a set of data structures (described in Sections 3 and 4) that can be used in all the different ways noted. This approach is not unknown in the hardware description language world --- ISPS has been used quite profitably in this fashion [2] --- but it is rare. We find that it offers sufficient leverage to be worth the additional effort.

12.2.2 Advantages

Our approach offers a number of features which, while not necessarily novel, do provide useful performance. For example, there is 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 behavior associated with that module (simulating the module in a single step), or "run the substructure" of the module, simulating the device according to its next level of structure. Enabling the behavior that spans the entire module gives us a one-step simulation;

enabling the abstraction shifting machinery that implements a port gives us a detailed simulation (by allowing information to propagate into the next lower level). Since the abstraction shifting behavior of ports is also implemented with the constraint-like mechanism described in Section 4, we have a convenient uniformity and economy of machinery.

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

Work in [6] described the importance of the "no function in structure" concept, essentially a point of methodology and "mental hygiene", suggesting that device behaviors be defined independent of their use in any specific circuit. We have adopted this perspective and built a small library of component descriptions (adders, wires, and-gates, etc.); a set of prototypical modules with structure and behavior descriptions independent of any particular circuit. Devices are then constructed by assembling and interconnecting instances of those prototypes, using the language described in Section 3.

A wire, for example, is a device whose behavior is a simple bi-directional propagation: information appearing at either one of its terminals will be propagated to the other. Any particular use of a wire typically has a single direction of propagation in mind, but our simulator runs the behavior description that reflects the "actual" (bi-directional) behavior.

While this approach offers no formal guarantees that the behavior definitions are free of implicit assumptions, it does provide an environment that strongly encourages attention to the issue by (i) distinguishing clearly between behavior definition of an individual module and its intended use in a circuit, and (ii) by forcing every module of a particular type to share the same behavior specification. This approach is especially 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.

Finally, our approach offers 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.

12.2.3 Limitations

The behavior specification mechanism we have described is quite straightforward and some of its limits are well-known. A set of constraints is, for example, 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. A bus protocol, for example, would require additional machinery to represent the state transition network describing the protocol. We might also want to describe and reason about behavior in higher-level terms like *enables*, or *inhibits*, suggesting the need for a vocabulary similar to the one developed in [25].

In addition, our current propagation mechanism works well when dealing with simple quantities like numbers or logic levels, but cannot deal with more elaborate symbolic expressions. 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 expression elsewhere. A simple but effective propagation of symbolic expressions is accomplished in [19], suggesting that the approach taken there may be a good starting point.

12.3 Limitations in Candidate Generation

This limitation in our propagation machinery is also responsible for the primary limitation in the candidate generation facility. As we have seen, the basic technique works by looking for a contradiction: we "turn off" the behavior rules for a single device and see if there is any set of assignments to its terminals that is consistent with the inputs and observed output symptoms. If no contradiction is reported, we consider the component to be a candidate.

But sometimes no contradiction is reported because the propagation machinery is too weak to discover it. One simple case can arise from reconvergent errors. In the simple circuit below (Fig. 20), for example, our current system is unable to determine that the input wire could not be a consistent candidate. When, in using constraint suspension, we "turn off" the behavior of the input wire, and insert the input (2) and observed output (7), several terminals end up with no values assigned. This is not considered a contradiction, so the input wire is put on the list of plausible candidates.

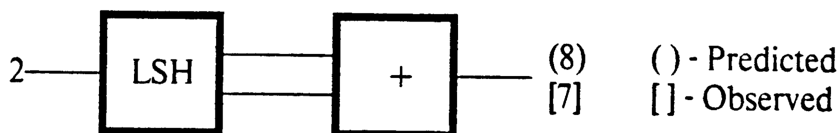

Figure 20 --- Reconvergent error. There is no way for the input wire alone to account for the symptoms. (The first device shifts its input left one place and sends the result to both outputs.)

In principle, we should have been able to rule it out, but including it as a candidate does not present any serious problems. The candidate set will be larger than it should be, so our system may end up doing some additional work in narrowing the set later (e.g., running more tests than should strictly have been necessary). The net result is some inefficiency, but no intractable difficulties. Enhancing the propagation machinery to handle this situation would involve difficult problems, including propagating and manipulating symbolic expressions [32], and reasoning about such properties as integer solutions to equations.

12.4 Comments on the Example

We chose a bridge example because they have traditionally been difficult; we considered the problem in TTL because of its common use. But this turns out to be a convenient choice. Bridges in TTL are easily modeled and the pattern easily checked for in the symptoms; we noted earlier the power this supplies in constraining the search space. In other technologies, unfortunately, the behavior of bridge faults is not so easily described. The reasoning is correspondingly more difficult and our system would be less focused in its generation of hypotheses.

A second problem highlighted by the example is the nature of the overall control of problem solving. Some parts are reasonably clear. The use of the categories of failure, for example, is to date at least fairly straightforward, starting with the most likely and moving toward the more exotic categories. We were also able to describe the overall strategy behind using and switching between multiple representations (using the functional as a hypothesis generator and the physical as a filter), but this is currently hardwired into the system. It would of course be better to have the system able to make this choice. Finally, we have yet to develop a globally defined strategy concerning the use of the description hierarchy. In general we work at the higher description levels before moving down, but this may not always be appropriate. It is not yet clear, for example, how "deep" to pursue candidate generation before stopping to generate distinguishing tests.

Consider the example of Fig. 7. There we determined that either the first multiplier or first adder could be at fault. Should we now drop down a level and try generating candidates inside the adder and multiplier, or should we try another test vector to provide more symptom data? Either of these strategies might indicate which of the two candidates contains the broken component. As noted earlier, dropping down a level may demonstrate that no subcomponent of (say) the adder can account for the original set of symptoms, hence it cannot be at fault: running a second test provides additional symptom data that may distinguish between the candidates. The appropriate strategy presumably depends on the costs in the specific case at hand: when we are very near the "bottom" of the description hierarchy, it may cost relatively little to go down the one final level; when tests are relatively cheap to generate and run, that may be the preferred approach.

12.5 The Example: Implementation

As we have noted, test generation in the example is currently done by hand, but all the rest has been implemented, in Franz Lisp running on a VAX 780. The example shown requires approximately 3 minutes of CPU time, but since this is early prototype, no attention has been given yet to producing efficient code.

The system is still a simple feasibility demonstration and as such lacks a number of design features necessary before it can be used as a serious tool. The current control structures, for example, are too deeply hardwired into the system. The sequencing through various categories of failure is currently embodied in a collection of procedure calls. Yet, given our emphasis on enumerating and keeping careful track of assumptions (as in Section 10), the selection of a failure category would more appropriately be accomplished with a general TMS system of the sort described in [15]. We do have a simple form of TMS in the dependency networks maintained by the simulation and inference rules, but it would be useful to construct a more general version and use it to keep track of the assumptions underlying the failure categories.

We are also working on a graphics interface that will allow dynamic display of the reasoning. As the sequence of figures shown earlier suggests, much of the process of candidate generation (for both the individual components and bridges) is easily understood in terms of diagrams. We are developing a system that will allow us to do this, displaying the results as they are generated.

12.6 Scaling: Device Complexity and Time

With any initial demonstration of this size, the scaling issue is always of concern. We have demonstrated the feasibility of a particular technique on a small combinatorial circuit assembled from simple devices. What happens when the circuit gets considerably larger and the devices get more complex?

We believe that size alone is unlikely to be a disabling problem, arguing that the design task imposes a limit on complexity. In order to make the design task tractable, a circuit with several hundred or several thousand components must have some sort description more compact than a simple listing of the components. Without such a hierarchy, it is unlikely it could have been designed successfully.

Complex behavior is likely to present a more difficult problem. Describing the behavior of devices at the scale of gates and adders is relatively straightforward. Describing the behavior of a disk controller is likely to be considerably more challenging, but, by the design argument once again, we speculate that it will not be overwhelmingly so.

The significant problem with complex devices is likely to lie in the inference rules. It is one thing to describe what a disk controller should do, it may be quite another task to infer what some of its inputs were, given its outputs. One important subproblem we will encounter here is propagating symbolic expressions, since, as noted, many devices are not uniquely invertible.

A second problem lies in extending our work to deal with more elaborate models of time and more complex devices with state. Our current system uses only the simplest model of time, enabling us to deal with simple synchronous devices. There are several steps we need to take to elaborate this. We have to model propagation delays so we can deal with races. Since protocols play an important role in communication, we need to be able to represent and reason about them.

Reasoning about devices with memory will also require elaboration of our candidate generation machinery. Our planned approach will be in the spirit of the current effort: where discrepancy detection currently moves us backward in space through the circuit, we intend to extend it to move backward in time as well, inferring values at previous time slices.

In facing the problem of scaling this feasibility demonstration up to problems of practical size, then, we find two issues of central concern: how can we generate inference rules for complex devices, and how can we model time in a way that allows us to reason over a significant number of slices? Work on both of these is currently under way.

12.7 Limits of Modeling: Analog Devices and Incomplete Models

Though we have set our sights here on reasoning from first principles, as with any representation we eventually encounter a level of detail not incorporated in our model. We have been working strictly in the digital world and as such cannot model or reason about analog phenomena. This is most obvious in the power failure example: our representation makes it easy to *incorporate* the insight that input ports can behave as output ports, but reasoning at the level of the digital abstraction precludes *deriving* that insight. Marginal signals are a second example: the effects might be captured at the digital level but some of the reasoning would be outside our current abilities.

Since every representation incorporates some level of abstraction, the issue is not that we encounter limits, but rather how important those limits are to the current domain and how difficult it would be to press beyond them. In answer to the first of these, we simply need more experience to determine how much of the problem can be captured at the level of detail we use.

Concerning the second, previous work (e.g., [13]) suggests that dealing with analog circuits and their continuous variables requires confronting significant additional problems. One difficulty arises because most analog devices are bi-directional and this often makes candidate generation considerably less constrained. There are more paths of interaction to be considered and hence more components in the circuit that could have caused the fault. In the example of Figure 9, for instance, allowing ports to be bi-directional widens the set of candidates (as it should) to include all the multipliers. Working in the digital domain means that most of our devices are uni-directional, keeping the set of candidate components smaller.

A second problem arising in analog circuits is the use of designs that rely on aggregate properties. The feedback or hysteresis in a device, for example, cannot be said to reside in or result from any one component. This produces difficulties when troubleshooting because some of the more interesting faults are those that are local to a single component but that disable the desired aggregate property. Tracing the fault from the disappearance of the aggregate property back to a single component can be difficult.

Finally, dealing with the continuous values present in analog devices presents its own set of problems, motivating much of the work on qualitative physics (e.g., [13]). We have to determine whether it will be possible to quantize the domain, developing and reasoning from a small vocabulary of labels like *high, low, float, rising,* and *falling*, or whether more complex machinery is necessary.

An equally pressing issue concerns the completeness of our models. Our system is focused in its efforts in part because its models of structure and behavior are complete: we knew, for example, exactly how the adder was constructed and how it behaved. Yet

much real-world troubleshooting is done with incomplete models. Experienced engineers employ much the same sort of reasoning shown here even when the device is a mainframe computer and the behavior is on the scale of an operating system, yet they clearly are not using complete models of either. The ability to specify and use incomplete models would thus help address the scaling issue and would increase the likely scope of utility of our work to fields like medicine, where complete models are simply unavailable.

12.8 But That's Not How It's *Really* Done

The performance of the program we have developed is in some ways noticeably different from the standard practice of a human expert. As expert systems work rather than cognitive science, the intent here is to be inspired by human performance without modeling it in detail. It is nevertheless useful to consider how our system differs from real practice.

While we argued earlier for the difficulty of troubleshooting based solely on empirical associations, it is clear that rules can serve useful roles. They might, for example, offer a form of memory to shortcut the process for problems previously encountered. This would clearly be an improvement on our current system, which will solve a problem from first principles every time, no matter how many times it is encountered. Such rules can also help focus the process by recognizing symptoms characteristic of particular kinds of malfunctions (e.g., power supply failures), characteristic of particular locations of failures (e.g., memory, I/O bus, etc.), or characteristic of particular machines (e.g., the disk controllers on this model tend to fail sooner).

Real troubleshooting also typically involves extensive use of logic probes, an issue this work has not addressed at all yet. While it will clearly be important in the long run, we would argue that it is appropriately delayed for several reasons. First, we claim that inference is "free", while measurement is often quite expensive. It's comparatively expensive in time: many inferences can be drawn in the time it takes to place a probe. More important, it's expensive in potential loss of information: it is often necessary to put cards on extenders or otherwise disturb the current state of the machine to make the measurement and the information lost can be crucial. The current trend in hardware speed makes this imbalance likely to continue.[18] As a result, we claim that it's well worth it to "think as much as you can" before taking another measurement.

The real issue here is the longstanding problem of information gathering. Where, contrary to our current assumption, we do not have complete data available, the real problem is reasoning from the current stock of information and deciding what measurement to take next. Our current examples are small enough that complete information is reasonable, but we will clearly encounter the issue as we scale the problem up to larger devices.

Finally, we might ask what fraction of the troubleshooting problem is not currently handled by simpler, existing technology of the sort characterized by stuck-at models and state of the art diagnostics. A significant percentage of the problem is solved in this fashion, but the fraction left unsolved turns out to be quite expensive. It is not unusual to find a strongly bi-modal distribution in which problems are commonly solved in either two hours or two days. There is thus a significant problem here, which, because of issues like decreasing design lifetimes, is likely to become worse.

---

18. There is some countervailing trend in the design of hardware that offers visibility of internal machine state and the possibility of automated probing.

13. RELATED WORK

13.1 Hardware Diagnosis

Two lines of work developed in the hardware diagnosis community have some interesting overlap with the work described here: the guided probe [5] and the effect-cause analysis of Abramovici and Breuer [1].

The guided probe technique has been in use in industry for some time and shares some basic ideas with our approach to troubleshooting. Both are based on the notion of discrepancy detection; both trace an error at an output back to its source by following the wiring of the circuit, and both use simulation to produce the correct values.[19]

One important difference arises because the guided probe approach does not have anything analogous to our inference rules (it uses a logic probe to measure voltages at nodes interior to the circuit). Having inference rules is important because it allows us to separate candidate generation from probing. That is, we can determine the entire set of plausibly broken components before making any additional measurements on the circuit. This can be an advantage because, given the entire candidate set, we may be able to select a few (or even one) place to probe that best reduces the size of the candidate set (e.g., the usual half-split strategy). The standard guided probe approach, by interleaving discrepancy detection with probing, in effect requires us to consider every candidate when it is first encountered.[20]

One further practical concern suggests additional utility of inference rules. Given the tendency toward increasingly exotic packaging technology, it is becoming more difficult to probe at random on a board. In such cases the inference rules become especially useful.

Finally, since the guided probe works directly from the schematic, it also inherits the fundamental problem noted earlier: the schematic may be wrong, and there is nothing in this approach capable of dealing with that problem.

Abramovici and Breuer have independently developed an approach to diagnosis that has a number of the features of a constraint-based system. Their deduction algorithm is similar to the use of inference and simulation rules in discrepancy detection, and they use multiple tests as we do to further prune the candidate set. One interesting result of their work is in its application to synchronous sequential circuits, where they show that this approach can diagnose a fault that prevents initialization (i.e., the initial states of the flipflops are unknown). This has long been a difficult roadblock for traditional fault-dictionary style diagnostics.

Drawbacks in the approach include its extensive use of inferences drawn from the fact that a wire can be labeled as "normal" (i.e., it has been observed to take on all possible values). The label is easy to establish in the binary world but clearly gets considerably more difficult for circuits modeled at higher levels. More seriously, the approach has been extensively explored for faults modelable as wires stuck at 1 or 0, but does not appear to go beyond this. One of the examples in [1], for instance, produces a unique diagnosis for a circuit under the stuck-at model, but does not indicate that, among other possibilities, a malfunctioning OR gate is an equally valid diagnosis. Finally, like other approaches that work solely from the schematic, it has no mechanism for considering that the schematic may be incorrect.

---

19. Some automatic testers use a board known to be good as the source of the correct values.

20. The lack of inference rules and corresponding need to probe suggests that, where we earlier characterized our approach as the interaction of simulation and inference, the guided probe is analogously characterized as the interaction of simulation and measurement.

13.2 Fault Models and Categories of Failure

In reviewing some of the traditional approaches to troubleshooting (Section 5), we noted several problems. We pointed out the historical trend toward using code originally designed for verification (proof that a device is totally free of faults) to do troubleshooting. We noted that code designed for verification requires fault models --- if we are to avoid exhaustive testing, we need to limit our tests to errors produced by a pre-specified list of faults. We then claimed that we do not need traditional fault models when the task is diagnosis and when a fault is defined as anything different from the correct behavior.

Yet in doing discrepancy detection we found it necessary to come up with an apparently similar sort of list: we had to enumerate the categories of failure in order to limit the paths of interaction we considered. Have we in fact made any progress or have we simply substituted one list for another?

We claim that progress has been made along two fronts. First, by focusing on troubleshooting and defining a fault as any discrepancy, we can deal with a wider range of faults, including any systematic misbehavior. Second, we have taken a step toward providing a somewhat more formal way of generating the entries on the list. The traditional fault model list is typically an informally generated listing of erroneous behaviors that have been observed in practice. Carefully examining the assumptions underlying our representation, as we did in Section 10, is in effect a "generator" of categories. It produces a number of well-known categories (e.g., local failures like stuck-ats, assembly errors, design errors) and suggests some less obvious ones as well (e.g., direction of information flow errors). While it is still not a formally complete generator, we at least have some relatively systematic basis for enumerating categories of failure to consider.

13.3 Other AI Work

As we have noted at several points above, a number of ideas developed in previous AI research have proved very useful in this undertaking.

13.3.1 Troubleshooting

From the work of Sussman and Steele on constraints, for example, we take the local propagation style of computation [32] and the maintenance of dependency networks [29]. Work by deKleer first demonstrated [11] that a first order theory of troubleshooting could be based on examining the dependency records left behind by a local propagator: any device on the path to a discrepancy should be considered a potential candidate; any device that participates in a "corroboration" (a place where predicted and measured values agree) can be ruled out. We used the first half of this approach in our candidate generator, when Step 2 of Fig. 8 used the dependency network leading to a discrepancy to determine the potential candidates.

Our work moves beyond this in two ways: by viewing troubleshooting in the framework of methodical relaxation of assumptions, and by the use of constraint suspension. Relaxation of assumptions handles bridge faults, while the approach in [11] specifically excludes all such errors in topology. The approach there works by examining dependency records produced by the local propagator, but the propagator worked from the original schematic. Nothing in this approach provides a way of entertaining the idea that the original schematic was incorrect; there is no mechanism for hypothesizing additional paths of interaction not shown in the original description.

Constraint suspension provides two additional advantages. First, like the approach in [11], it determines which components might be at fault, but then it also determines symptom values for the candidates. This provided important information in solving the bridge fault problem in particular, and in general allows candidate generation to continue at successively lower levels of description.

Second, as noted in [11], the use of corroborations in the first order theory above runs into trouble with any device whose behavior can inhibit propagation of an error (e.g.,

an AND-gate or multiplier whose other input is 0). Consider for example a slight modification to Fig. 7. If ADDER-2 were a fourth multiplier and input E were 0, we would get a corroboration at G (0 expected and observed), apparently exonerating MULT-2. But in that circuit MULT-2 would in fact be a valid candidate. Our constraint suspension approach handles this situation without difficulty: by running the behavior descriptions of all but one component (the one being tested for candidacy), it effectively determines the appropriate consequences of all discrepancies and corroborations.

The approach in [11] was developed further in SOPHIE [6] to include knowledge of component fault modes (e.g., resistors can be shorted, open, high or low), and to include knowledge about higher-level modules that was specific to the particular circuit and module. Our approach differs in using a uniform approach to device modeling and troubleshooting, where SOPHIE's troubleshooting of the high-level modules was handled using the circuit specific knowledge (the first principles approach was used at the level of primitive components).

That system also noted the potential difficulties arising from implicit assumptions, and speculated about the use of a general truth maintenance system to deal with the problem. This does not appear to have been implemented.

We have drawn from the work in [6] in some other respects, both for specific techniques and overall approach. That work demonstrated the use of simulation as a fundamental tool in troubleshooting (using it for example to predict observations from inputs and to predict consequences of failures). It showed how dependency chains could support hypothetical reasoning. The work also made clear the importance of "mental hygiene" in building simulation models. Ideas like the "no function in structure" concept were both directly useful (as noted in Section 12) and helped sensitize us to the importance of implicit assumptions.

That work also kept distinct the kinds of knowledge used in doing troubleshooting. The system relied on dependency records to generate the candidate set first, and only then used knowledge about component fault modes to prune the set.

13.3.2 Adjacency

The notion of adjacency has been pursued from several different directions, some of which proved useful in our own approach to the concept. The original conception of constraints [32] helped define the issue and characterize local propagation as a way of thinking about computation. Work by deKleer [12] and deKleer and Brown [13, 14] developed a number of principles involving local propagation when reasoning about cause and effect in physical devices.

A slightly different conception of adjacency underlies some of the work of Lenat [20]. There, syntactic changes to LISP descriptions produced useful new mathematical concepts because LISP and mathematics are languages that are "close" together. That is, the primitives and method of composition are similar enough that the languages are structurally similar: concepts that are "adjacent" mathematically often have LISP definitions that are quite similar. As a result, making small syntactic changes to a LISP expression that captured one mathematical concept often produced an expression embodying another meaningful mathematical concept. This underscores the importance of the choice of representation language when making changes to a description: the changes will produce meaningful results to the extent that the languages share a definition of adjacency. A similar inspiration lies behind our analysis of the need for the "right" representation in understanding different kinds of faults: the physical definition was appropriate for bridge faults because it provided the appropriate definition of adjacency.

In Section 11 we suggested that it can be useful to have several different definitions of adjacency, provided by the multiple representations (functional, physical, etc.) employed there. A related notion, multiple views of a circuit, shows up in the concept of slices [31] used in design. A central observation in that work was that simplifying the design task requires being able to have several different views of a circuit, each one

packaging up things differently. The basis for the packaging is typically teleological: a particular section of a circuit is packaged up in a slice because it has an identifiable behavior that accomplishes a particular purpose. The design task can then be simplified by reasoning about the device using that abstraction, without being concerned about how that behavior is actually accomplished. Multiple slices can provide multiple views of the same section of the circuit; slices thus offer a mechanism for expressing multiple abstractions.

Our emphasis has been on using different definitions of what adjacent can mean in order to produce fundamentally different representations for troubleshooting. Slices may offer multiple different views of the circuit, but they are all functional views and share the same basic representational vocabulary (that of functional modules). We have both a functional representation with its definition of adjacency and its vocabulary, and a physical representation with its own distinct definition of adjacency and vocabulary (cabinets, boards, chips). Our focus is thus not on having multiple views of the device, but on having fundamentally different representations, and on carefully enumerating the different criteria (definitions of adjacency) that can be used as the basis for each representation. The definitions are in turn derived from the different pathways of interaction, since those pathways reflect the mechanisms by which faults manifest.

Finally, our categories of failure are similar in spirit to the class wide assumptions in [13]. As pointed out there, it is impossible to write assumption-free descriptions of behavior. The problem manifests itself in the most basic terms in simply choosing a vocabulary. Any behavior description will be built from a finite vocabulary of terms, yet there is no obvious limit to the set of terms that *might* prove relevant (weight, color, flexibility, etc.). Since we have to choose a finite vocabulary, we should at least provide an explicit, defensible set of selection criteria. The problem arises also in using a given vocabulary to describe a device. In describing a door bell, for example, the model in [13] needs to refer to the effect of the coil's magnetic field on the clapper, but wish to invoke the common assumption that the field is not strong enough to induce currents in nearby wires. Again we need explicit, defensible grounds for making such assumptions.

The authors suggest that introductory physics provides a convenient and routinely, if tacitly, used set of criteria; they term them "class wide assumptions". These are assumptions about behavior whose utility (and credibility) arises in part because they apply to whole sets of devices. Ignoring the effect of the magnetic field on nearby wires, for example, is a reasonable assumption because it applies to a wide class of devices.

Our categories of failure are similar in spirit. We are motivated by the same original consideration, namely that it is impossible to create assumption-free descriptions of structure and behavior. We have made the identification and handling of such assumptions a primary concern and have been able to provide a somewhat more systematic generation of the categories. Since deKleer and Brown are concerned with qualitative physics, they appropriately look to that field for inspiration and have the task of identifying and accumulating the assumptions. Our more tightly focused concern --- troubleshooting --- allows us to examine what kinds of things can go wrong, and makes possible the relatively systematic generation of assumptions accomplished by perturbing the representation, as we did in Section 10.

This also reflects our concern with enumerating the pathways of interaction as completely as possible. While class wide assumptions reflect important knowledge about the domain, that knowledge is often in the form of a reason not to include a particular pathway of interaction (e.g., omitting the path from the coil to nearby wires in the door bell example). We want to press one step further on by building a list of possible pathways that is as systematic as possible, and then consider the variety of fault characterized by the existence (or omission) of such a pathway.

14. SUMMARY

We seek to build a system that reasons from first principles in understanding how devices operate. We find troubleshooting of digital hardware to be a tractable and fertile ground for exploration. We have developed languages describing structure and behavior that distinguish carefully between them, and provide information about structure that is organized both functionally and physically.

We find that the traditional machinery for troubleshooting focuses primarily on test generation and its use in verification of device behavior. Our problem is better characterized as diagnosis in the presence of known symptoms, in devices complex enough that it is important to use the symptoms to help guide the troubleshooting.

We view the process as the interaction of simulation and inference, with discrepancies between them driving the generation of candidates. Discrepancy detection and tracing through dependency records gives us a foundation for troubleshooting that identifies potential candidates. The technique of constraint suspension extends this by supplying symptom values for the candidates and handling both discrepancies and corroborations with a single mechanism.

In exploring this approach further, we find that the concept of paths of causal interaction plays a key role, supplying the knowledge that makes the machinery work. We need an explicit model of causal interactions in order to determine which components to consider. The amount of such knowledge then leads us to a fundamental dilemma: the desire to deal with a wide range of faults seems to force us to choose between an inability to discriminate among candidates and the inability to deal with some classes of faults.

In response we have developed a troubleshooting strategy based on the methodical enumeration and relaxation of underlying assumptions about the device. We were able to generate the assumptions in a relatively systematic fashion by examining the module and information path representation of hardware. By considering the consequences of violating each assumption, we were able to generate a collection of categories of failure.

We then invoked a version of Occam's razor, noting that some categories of failure are more likely than others. This provides a criterion for ordering the categories and pathways of interaction to be considered. We start with the simplest category of failure first and consider only one class of paths of interaction initially. If this fails to generate a consistent hypothesis, we surrender one of our underlying assumptions, adding the next category of failure, and consider additional pathway of interaction.

We illustrated this approach by diagnosing a bridge fault. When our initial categorization --- local failure of function --- encountered a contradiction, we surrendered the assumption that the schematic was correct and considered one additional path, bridges. This staged relaxation of assumptions permitted a constrained generation of hypotheses. Within the bridge fault category, additional restriction was then provided by using constraints associated with both the physical and functional representations.

Drawing back from this specific example, we explored several possible generalizations of the work. We found applicability in software, for example, for the notion of enumerating the assumptions in the representation and using this to generate categories of failure. While the particular pathways of interaction were different, the technique appears to phrase a relevant set of questions. We found that some errors in software are usefully thought of in terms of a set of pathways of interaction from that domain.

Our overall approach also suggests that part of the expertise of a domain lies in knowing how to simplify a problem, i.e., knowing what simplifying assumptions can be made, recognizing when an assumption has failed to help, and knowing how to recover and get the solution back on course. Enumerating and ordering the categories of failure provides one simple mechanism for expressing such knowledge.

A further generalization of this work came from examining the difficulty involved in dealing with bridge faults. We found that an important property of a representation is its definition of adjacency. In this view bridge faults are difficult because they are simple and local changes to the physical representation, but neither simple nor local in our original,

functional representation. We started with the functional representation because we were presented with behavioral manifestations of a fault, and hence needed to reason from a representation organized according to behavior. But this proved to have an inappropriate definition of adjacency for the problem at hand, so we shifted representations. This in turn lead us to a useful guideline in choosing representations for diagnostic reasoning: we should attempt to find a representation in which the suspected change can be viewed as a compact modification affecting adjacent devices.

The underlying rationale for focusing on defining adjacency is a belief that faults manifest through processes that act locally, i.e., there is no action at a distance. We then found that we could define a number of useful kinds of adjacency by considering the paths of interaction. Each of these defines a different metric (Euclidean, thermal, electromagnetic, etc.), each generating a different representation.

We pursued this one additional step, asking why we believe there will in fact be some representation in which the fault can be seen as compact. The belief appears to rest on the heuristic that a malfunction in a previously operational device often results from a single cause rather than a number of independent events. Like any heuristic, this may not always be correct, since multiple, independent failures do occur. But it is true often enough and there are substantial advantages to choosing a representation with a good definition of adjacency.

Because our basic representation machinery employs little more than the traditional black box notion of information transmission, we suggest that some of the central concepts in this work may have relevance of considerable breadth. The examination of assumptions underlying the representation and constraint suspension, for instance, appear to be applicable in a number of different areas. We noted above the potential use in software of examining underlying assumptions. We conjecture that both techniques may apply to to any system that might be modeled in terms of information transmission, ranging from hardware, to software, to organizations. Organizations may have pathways of interaction other than those on the personnel chart, for example, and we might consider "organization troubleshooting" via constraint suspension.

Finally, we observe that there has been growing focus on the power contributed by choosing a "good" representation and the utility of multiple representations. We suggest that one of the characteristics of a good representation is that it provides a definition of adjacency that makes the fault appear compact, and that the utility of multiple representations arises in part from the multiple different definitions of adjacency they provide.

Acknowledgments

Contributions to this work were made by members of the Hardware Troubleshooting project at MIT, including: Howie Shrobe, Walter Hamscher, Mark Shirley, Harold Haig, Art Mellor, John Pitrelli, and Steve Polit.

Some of the initial inspiration for this work came from conversations with Ed Feigenbaum; periodic arguments with Mike Genesereth have helped sharpen vague intuitions and were an early source of specific examples. The presentation in this paper was improved by comments from John Seeley Brown, Johan deKleer, Mark Shirley, and Patrick Winston.

REFERENCES

[1] Abramovici M, Breuer M A, Fault diagnosis in synchronous sequential circuits based on an effect-cause analysis, *IEEE Trans. on Computers*, Vol. C-31, No. 12, December, 1982, pp. 1165-1172.

[2] Barbacci M R, Instruction set processor specifications (ISPS): The notation and its applications, Carnegie Mellon University Technical Report CMU-CS-79-123, May 1979.

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

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

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

[6] Brown J S, Burton R, deKleer J, Pedagogical and knowledge engineering techniques in SOPHIE I, II, and III, in Sleeman and Brown, eds., *Intelligent Tutoring Systems*, Academic Press, 1982.

[7] Davis R, et al., Diagnosis based on structure and function, *Proc AAAI-82*, August 1982, pp. 137-142.

[8] Davis R, Reasoning from first principles in electronic troubleshooting, *Intl Journal of Man-Machine Studies*, 1983, 19:403-423.

[9] Davis R, Buchanan B G, Shortliffe E H, Production rules as a representation in a knowledge-based consultation system, *Artificial Intelligence,* 8:15-45, February 1977.

[10] Davis R, Shrobe H E, Representing structure and behavior of digital hardware, *IEEE Computer*, Sept. 1983, pp. 75-82.

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

[12] deKleer J, The origin and resolution of ambiguities in causal arguments, *Proc 6th IJCAI*, August 1979, pp. 197-203.

[13] deKleer J, Brown J S, Assumptions and ambiguities in mechanistic mental models, *Xerox PARC Report CIS-9*, 1982.

[14] deKleer J, Brown J S, Naive physics based on confluences. Xerox Parc Report, 1983.

[15] Doyle J, A truth maintenance system, *Artificial Intelligence*, **12**, 1979.

[16] Estrin G, A methodology for design of digital systems --- supported by SARA at the age of one, *Proc NCC*, 1978, pp 313-324.

[17] Gaschnig J, Preliminary evaluation of the performance of the PROSPECTOR system for mineral exploration, *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 308-310, 1981.

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

[19] Kelly V, Steinberg L, The CRITTER System - Analyzing digital circuits by propagating behaviors and specifications, *Proc AAAI Conference*, pp. 284-289, August 1982.

[20] Lenat D, Heuretics: theoretical and experimental study of heuristic rules, *Proc National Conf on AI*, 1982, pp 159-163.

[21] Lim W Y-P, HISDL --- A structure description language, *CACM* 25:823-830, Nov., 1982.

[22] Newell A, Simon H, *Human Problem Solving*, Prentice Hall, 1972.

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

[24] Pople H, Heuristic methods for imposing structure on ill-structured problems, in Szolovits (ed), *Artificial Intelligence in Medicine*, AAAS Selected Symposium 51, 1982.

[25] Reiger C R, Grinberg M, A system for cause-effect representation and simulation for computer-aided design, in Latombe (ed.), *Artificial Intelligence and Pattern Recognition in Computer-Aided Design*, North-Holland, 1978, pp. 299-334.

[26] Roth J P, Diagnosis of automata failures: a calculus and a method, *IBM J Res. and Devel*, pp. 278-291, 1966.

[27] Shirley M, Davis R, Digital test generation from hierarchical models and symptom information, *Proc. IEEE International Conference on Computer Design*, Nov. 1983.

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

[29] Stallman R M, Sussman G J, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence*, 9:135-196, 1977.

[30] Steele G, The definition and implementation of a computer programming language based on constraints, MIT TR-595, August 1980.

[31] Sussman G J, Slices: at the boundary between analysis and synthesis, in Latombe (ed.), *Artificial Intelligence and Pattern Recognition in Computer-Aided Design*, North-Holland, 1978, pp. 261-299

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

[33] vanCleemput W M, An hierarchical language for the structural description of digital systems, *Proc 14th Design Automation Conf*, pp 377-385.

[34] Winston P, *Artificial Intelligence*, Addison Wesley, 1984, second edition.