

## AI Approaches to Troubleshooting

Dr. Johan de Kleer  
Xerox Palo Alto Research Center

I would like to point out that this talk represents work done with John Seely Brown. Like any good AI expert, I have an enormous pile of slides so I can dynamically adjust my talk depending on the necessity of the situation. I'm very tempted to give a talk today which I won't give, but it involves why AI is bad for you. It really is, but this is probably not the right forum for it.

Let me start with basics. Why are we here? The single common thread is that we have technological artifacts and they break faster than we can fix them. From that keen insight, one can make further observations about the two ways to fix broken machines--you get people to fix them or you get other machines to fix them. Right now, both of these approaches don't work very well. Is there a common theory of troubleshooting underlying these two approaches? To put the question another way: Is the theory of troubleshooting underlying human training and human aiding and doing maintenance any different from the theory underlying computer-based test equipment? The argument is often made that there is a fundamental difference because inference is cheap for computers. This, however, remains to be seen.

In this talk I'm going to take as an objective getting the best computer-based troubleshooter we can. What you'll see at the conclusion of my talk is that, although I never took it as a goal, the design that I come up with is something that is just right for the training problem.

We know that machines are hard to fix. However, we also notice that some people are good at fixing them, and we call those people experts. We talk to those experts for awhile and we discover that they use knowledge about circuits to diagnose. It takes intelligence to find faults. Now you will notice that any good data base indexing system would come up with a hit on those three key words I've just said. What is it? The hit is "artificial intelligence." (This, by the way, is the beginning of my 45 minute anti-AI talk.) The piece of reasoning I just went through about expert-knowledge-intelligence being equated with artificial intelligence is completely fallacious. If it were true, any problem in our society for which there exist a few experts and which seems to require intelligence would be placed in the arena of AI. And, by that definition, AI would include everything. Now I know lots of my friends in AI believe that, but it's completely false. You may be able to guess the other 44 minutes of what I would say about the dangers of AI.

Another story I'd like to tell you involves picking someone at random, say, a knowledge engineer. The knowledge engineer often uses a sledge hammer to crack a very small nut--a far too large sledge hammer. But perhaps it's a case of the nut cracking the sledge hammer. The point is that AI is hairy: it's hard, complicated, and usually doesn't work. Now the problem is that it is also sexy. I want to drive home a simple piece of engineering methodology that we should all

be completely familiar with, but somehow is often forgotten--you don't want to do the intelligent thing, you want to do the simplest thing that works. If you don't do that, you're going to get into trouble. I am not in the business of building a troubleshooter that's going to work in the field in 2 years, so I have a different goal than some of you and have to make different kinds of decisions. If I had to build a troubleshooter to work in 2 years, I would do different things than I am doing now. The kinds of questions I'm interested in include: What is troubleshooting? What is the fundamental problem? What does it mean to understand a machine? Nevertheless, this little piece of methodology that I've described above applies to the science as well as to the engineering. If I propose a complex mechanism or artifact, I better have a reason for why I introduced the complexity. Whether it be engineering or science, do the simplest thing that works and have an explanation for any complications in the product.

In the rest of this talk, I'm going to get a little more technical and lay out my perspective on troubleshooting. I'm going to have to do that while I'm going through some examples. I won't go very deep, but I hate talking in completely high level terms.

Let's consider the regulated power supply circuit shown in Figure 1. I'm going to talk about six approaches to troubleshooting using this circuit. Some of them I have implemented; some of them are fictional. I'm presenting them to demonstrate the spectrum of possible AI approaches.

First of the six approaches to troubleshooting is the modern approach, which is, I gather from some of the other talks I've heard at this workshop, now the conservative approach. The second approach is the empirical association approach. The third approach examines organization of knowledge along structural and causal lines. Fourth, I'll describe the approach that I consider the most powerful--how to use deep knowledge about behavior to make troubleshooting inferences. Fifth, the use of deep knowledge about fault modes, and sixth, the use of causal models. Causal models is actually a separate talk so I won't go very deeply into that subject.

As we go through this list of six approaches to troubleshooting, we gain certain advantages with each successive approach. Thus, if we actually want to build something soon, we have to take a tradeoff someplace and choose a position along the spectrum. The way to view this framework is as a tradeoff of knowledge vs. inference. There's a disastrous slogan going around that states "knowledge is power." Actually, we want as little knowledge as possible and we want our inference to be as powerful as possible. The basic idea is that if we have a little knowledge it's easy to change, easy to debug, and takes less effort to make sure that it's right. So, as I go through these six approaches to troubleshooting, I'm going to lay out where the knowledge is, how it is used, and what it is good for.

Let's begin by discussing what a troubleshooter does. Superficially, the troubleshooter seems to do two things--makes a bunch of measurements, then replaces a component. It sounds very simple. But note that in the activity of measuring, the troubleshooter is presumably doing two things. He is making measurements from which he computes entailments about the components of the device and based on those entailments, he decides to make yet other

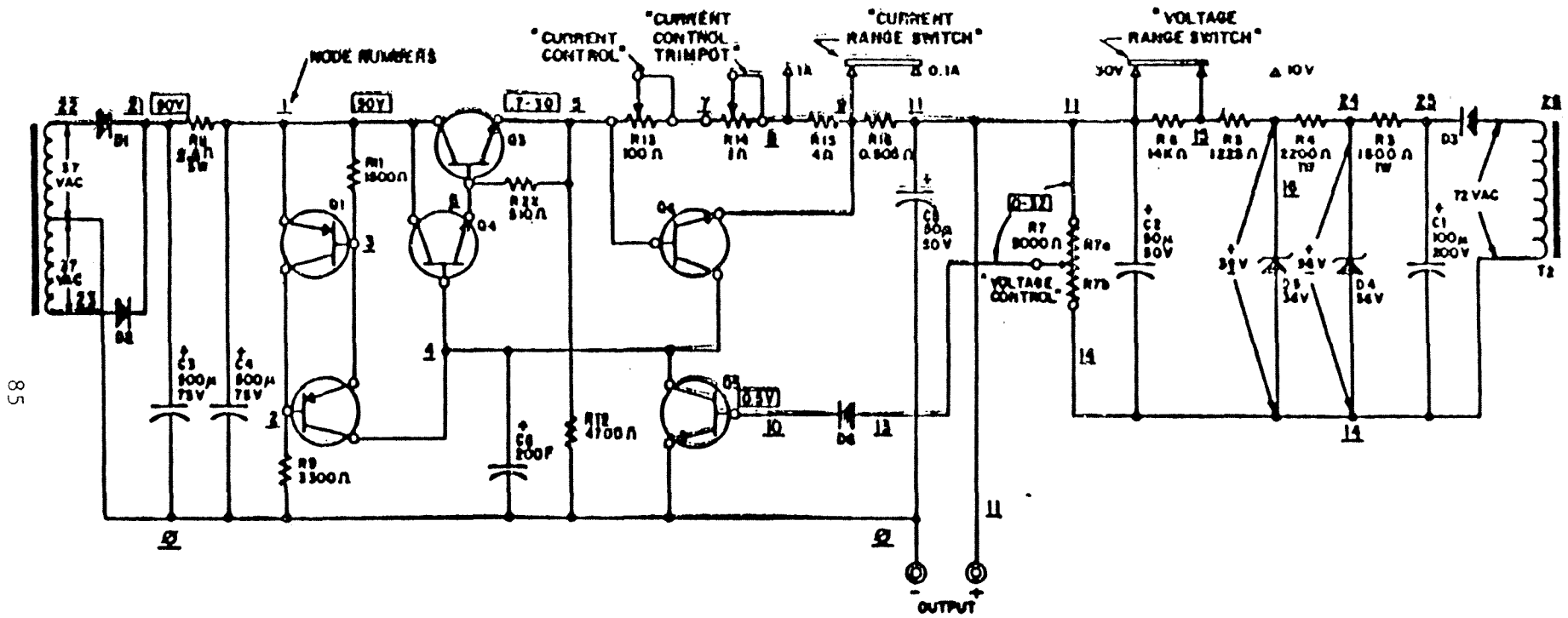


Figure 1. Schematic of the IP-28 Power Supply.

measurements. The point is, he wants to make the measurement that has maximum information gain.

What goals do we have for building a troubleshooter that has performance? There are four. The first one is robustness. We want the troubleshooter to succeed on faults that the designer of the system did not foresee. We do not want unexpected inputs or unexpected faults to blow the troubleshooting system out of the water. Second, we want it to be general and work on a wide number of cases. It should be able to apply to other circuits besides the one for which it was implemented. Third, we want it to be efficient. Anyone can build troubleshooters that make a measurement at every node and eventually figure out what is wrong. But measurements are expensive. Some are more expensive than others, so there is some metric within which a troubleshooter optimizes. The fourth goal is constructibility. It has to be easy to build the troubleshooter; if it takes 5 years, then something is wrong.

There is a fifth goal for building a troubleshooter that I have purposely not included on my list--explanation. Explanation is often oversold. The reason explanation is such an issue for expert systems is that the implementers of expert systems have to convince the other researchers in this field, and their own managers, that these programs work when in actual fact they don't. An explanation facility provides a means for failing gracefully. It makes the system appear as if it might work and deflects questions from the real issue which is performance. If the conclusion of the expert system was always right, or mostly right, you wouldn't need an explanation, you'd just follow it. And the second point is, in many contexts, the technician that's using the expert system doesn't have the technical training to evaluate the explanation that the expert system would generate anyway. So explanation is oversold in this business.

Let me quickly outline the modern approach, although I'm sure many people at this workshop have given talks about it. As far as I can tell, the modern approach is to write 100,000 lines in Pascal code for your million dollar piece of diagnostic equipment which finds 30 percent of the faults in the device. Let's evaluate this approach using my four criteria. First of all, look at the knowledge in the system. The knowledge is all implicit in the code. There is no inference going on at all. The first problem is that presumably the writer of the Pascal program went in with a set of faults that had to be covered. How can you tell the Pascal program actually covered the faults? Secondly, how do you know that the original set of faults were the faults that were important? So the robustness of this 100,000 lines of Pascal program is pretty bad. It's also not general at all. There's no hope this Pascal program is going to work on another circuit. Regarding efficiency, it seems many of the programs I've heard about are very inefficient in the sense that they make the same set of measurements no matter what. I've heard of frustrated technicians who have to wait hours for the program to reach the test of that part of the modules they suspect is broken. But the Pascal code doesn't know when to start and when to stop. It just runs through all of the tests. Our fourth goal of constructibility is obviously poor for the modern approach.

So how can we do better? Well, we go observe and talk to real-life troubleshooters. The human expert says things to us that resemble "if-then"

relationships. For example, "if the voltage is low, then the constant current source is open." Aha! These are called empirical associations abstracted from the expert. So we start writing these empirical associations and they work! So we collect a very large number of rules. Now, this is a little oversimplified, but I'd like to point out this simplified system actually works because empirical associations are useful for both of the troubleshooter's basic tasks: computing the entailments of measurements and proposing new ones.

Suppose the troubleshooter has a rule that says: "If V5 is low, then R9 is open." The troubleshooter then measures voltage at node 5. Suppose he discovers that this voltage is low. From that he can deduce by simple antecedent reasoning that R9 is open. So the rules allow the troubleshooter to compute the entailments of his measurements.

These rules can also be used to propose measurements. For example:

If V5 = low, then R9 is open.  
If I3 = high, then [R1, R9, C6] is bad.  
If I9 = low, then [R4, Q4, T1] is bad.  
.  
.  
.

First, pull out all the rules whose consequences mention components still in the candidate set. From those choose an antecedent which is still unknown, but which if known, would provide maximum information about the remaining components in the candidate set. Maximum information gain is a function of the cost of measuring the antecedent and how well it works to split the candidate space in half. In the above example, we would measure I3.

What are some of the problems of using this approach to troubleshoot? The main problem is illustrated by the knowledge engineer trying to fit empirical associations into the knowledge base. I'd estimate that even a simple circuit like the IP-28 requires something like 10,000 rules. So, what happens when you have to add the 10,001st rule? Where do you put it? Well, you just don't know. The rules are flat and unstructured. How do you know the new rule doesn't contradict previous ones? How do you know that you haven't missed some set of faults? It's almost impossible to tell, and that's the intrinsic fatal flaw of any troubleshooting scheme based on empirical associations.

There are advantages to this approach, however. One is that the knowledge is now explicit in the program. It's also a bit easier to modify and we've got some inference, although it's very weak. However, its robustness is still suspect because there are 10,000 rules that had to be written down. The generality is low, but it's better because 10,000 rules are better than 100,000 lines of Pascal. Efficiency is actually good because the measurements that it proposes depend on previous measurements. In fact, it comes close to being optimal. Constructibility is still bad, however, because extracting 10,000 rules is a very painful task.

Before I move on to more complicated stuff, let's look at a very simple idea that gets around many of the problems associated with the empirical association method. Consider the pool of rules. We notice that the pool can be broken into subsets associated with circuit modules. That observation is based on the fact that the function of a circuit arises out of this structure. The device has a physical manifestation. This structure can be used to organize the rules.

Let's say the knowledge base contains rules which state that if V6 is low, this implies that Q6 is all right or "OK." If V6 is low, that implies that Q5 is OK; and if V6 is low, that implies that R6 is OK. If a troubleshooter incorporates an explicit notion of structure, these three rules can be collapsed into one rule: If V6 is low, this implies that V (ref) is OK. This idea is very simple, but it actually gives you a surprising amount of leverage. For example, if you see that the same antecedent says something about 3 of the 4 components of a module, maybe there's a rule missing for the fourth component.

Similarly, a causal model like that of Rieger and Grinberg can be used in the same way to organize the rules. In fact, it really doesn't matter much what is used to organize the rules. If there's only a little bit of truth in what you use to organize rules, the robustness is going to be improved as it is easier to notice missing rules.

The three approaches to troubleshooting we've discussed thus far have ignored the fact that the device operates as a consequence of the behavior of its parts. That's a radical observation for the people in the empirical association camp. The idea of the fourth approach to troubleshooting, the use of deep knowledge about behavior, is very simple. Consider the example in Figure 2:

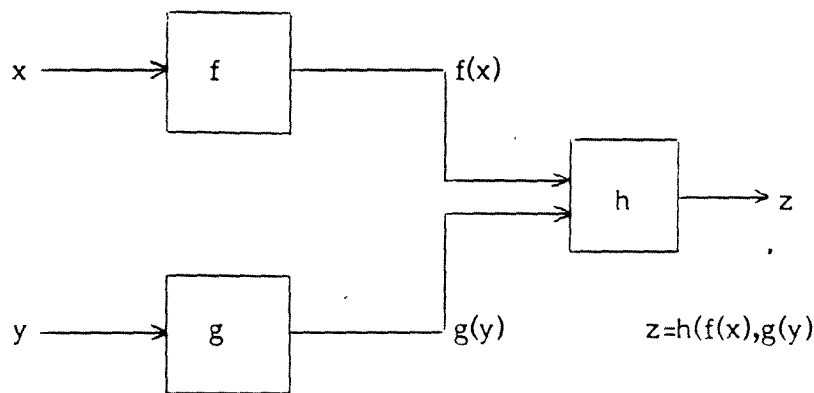


Figure 2.

There are two inputs to the device:  $x$  and  $y$ .  $x$  goes through module  $f$  causing  $f(x)$ ,  $y$  goes through module  $g$  causing  $g(y)$ , and both are inputs to module  $h$  causing output  $(h(f(x),g(y)))$ . This sort of reasoning allows us to compute an expectation of what the value at  $z$  must be if we know what are the inputs. Then we can make a measurement and see if it differs from expectations. Let's suppose that  $z$  measures differently than expected. One possibility is that  $h$  is faulty. Or, we can assume  $h$  is OK and  $g$  is correct and compute what the faulty  $f(x)$  is. Knowing  $x$  and the faulty  $f(x)$  we can compute the fault in  $f$ .

That's not a bad idea, but it won't quite work. It might work for simple cases like this one, but it won't work in general. Often this algorithm will impugn every component of the circuit. The reason is that if the circuit contains memory or feedback or is analog or if the wires are bidirectional or the functions are hard to invert, the method won't work. In other words, we are going to have to do something different, because I've just covered 99 percent of the circuits in the world.

What we should do is describe the behavior of every component by a constraint that can be used in any way--forward, backward, or sideways. There's no notion of forward and backward in constraints. An example of constraint is Ohm's Law  $E=IR$ . If you know the voltage and the current, then you can determine the resistance. It doesn't cause the resistance. If you know the resistance and the current, you can determine the voltage, but the resistance and the current don't cause the voltage.

We can use the constraint propagation, the idea of composing constraints with each other, to construct expectations. In Figure 3, suppose you measured the voltage between node 15 and node 14 and node 16 and node 14. Using Kirchhoff's Voltage Law, you can deduce the voltage across resistor R5. If you know the voltage across the resistor, you can deduce the current through it using Ohm's Law of constraint. This leads to an expected R5 current of 3 milliamperes (mA). However, that's only valid assuming that R5's resistance hasn't shifted from what it should be. If R5 has shifted in value, the predicted and measured currents are going to be different. Thus, R5 is an assumption of the propagated current of 3 mA.

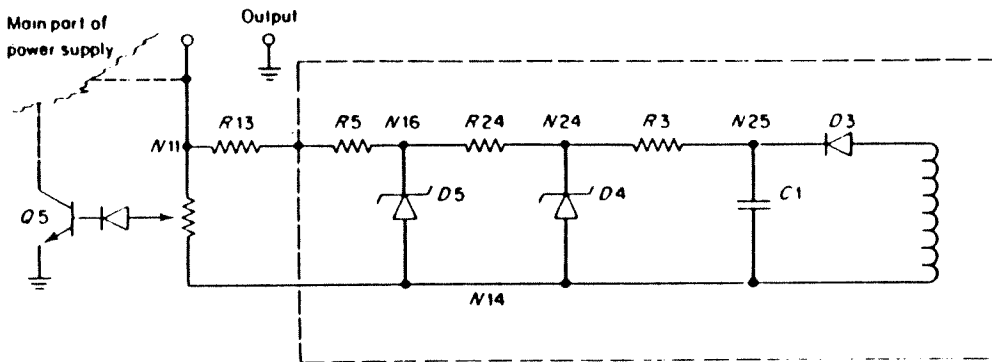


Figure 3. Constant voltage source

Moving on, the voltage across D5 was measured to be 34 volts which is less than the zener's breakdown voltage. Thus the current through D5 is zero. Notice that this propagation depends on two assumptions: (1) R5 is not faulted and (2) D5 is not faulted. Using Kirchhoff's Current Law we can propagate this current through R4. As I'm assuming there are no extra wires or open wires, this propagation step adds no new assumptions. Knowing the current through R4 allows us to propagate the voltage across it by Ohm's Law. Of course, this propagation is dependent on R4 still having its expected resistance. Thus, the

voltage across R4 is 7.18 volts assuming that R5, D5, and R4 have not shifted in value.

Now, how do we use this information to troubleshoot? The basic idea of deep knowledge about behavior as an approach is that the propagations allow you to construct expectations of circuit behavior. If you construct an expectation and make measurements at a point of expectation, you get some information. There are two things that can happen. On the one hand, the measurement can corroborate the expectation. For example, we might measure the voltage across R4 and discover it actually is 7.18 volts. Thus, the fault does not lie in R5, D5, or R4. On the other hand, if we discover that the voltage across R4 is zero volts, the measurement disagrees with the propagation. We would then conclude that either R5, D5, or R4 is faulted. Furthermore, if we assume that the circuit contains only one fault, every other component in the circuit is unfaulted.

This troubleshooter proposes new measurements by choosing those expectations to corroborate or conflict. It examines the set of all expectations that have been constructed for which a measurement has not yet been made, picks the one which gives maximum information, and then makes that measurement. In other words, we have a troubleshooter that both computes entailments and proposes measurements.

Lest you think that I am completely out in a dream world with this simplistic scheme, let me illustrate a few of the things that have to be patched to make the propagation-based troubleshooter actually work. In the real world there are meter errors, manufacturing errors, and wide tolerances on transistor specifications. Therefore, the troubleshooter has to consider the variability, that's problem number one. Second, I know that component models aren't all constraints. The models are made up of inequalities on voltages and currents. So actual component models are fairly complex.

The second reason you might think that I'm wrong is that the corroborations and conflicts all go out the window when the values have tolerances. For example, it gets far more complicated when comparing expectations with measurements. In general, you have four cases. First, if the ranges of the expectation and the measured don't overlap, you've got a conflict unambiguously. Second, if they corroborate and they're roughly equal, you actually do not have the necessary evidence to rule out the assumptions underlying the expectations. This corroboration could be correct, yet the underlying components could still be faulted because there is so much variability in the values. Another more sophisticated mechanism is used to solve that problem.

Third, what happens if the measurement is very tight but the expectation has a very broad range, or the reverse? For each of these cases, information can be gathered about the candidate set.

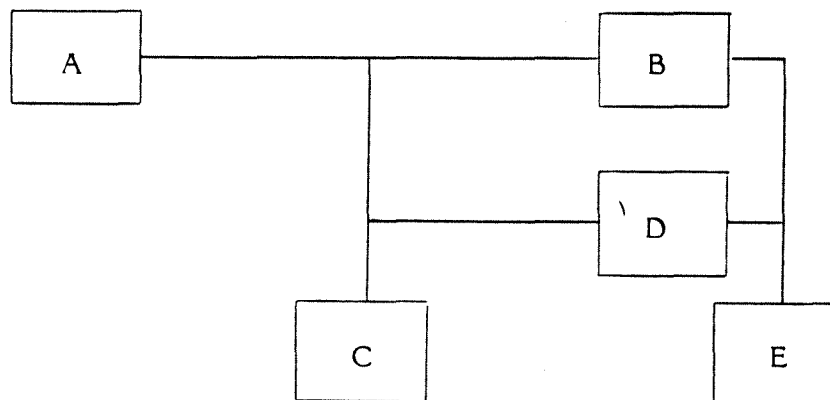
So far we are constructing expectations that are not causal, we are merely using rules about how components behave in general. Now, one of the advantages of this new approach is that all of a sudden all the necessity for most of the empirical associations goes away. Most of the rules get automatically

handled by the coincidence mechanism. The propagation-based mechanism directly obviates the need for most of the rules by instead making inferences about the faultedness of circuit components based on first principles.

Suppose, for example, we had a rule which said if the voltage across R5 was low, the CCS was open. Under the old system, another rule is needed which states that if the current through R5 was low, the CCS is open. In the propagation-based scheme, this second rule is now superfluous. If you discover the current through R5, the propagator will construct the expectation that the voltage across it is low, and that triggers "voltage low." (That expectation is based on R5, so that antecedent assumption must also be included in the consequent of the rule.) As a consequence, the rule set collapses dramatically. In fact, by this point, we have reduced the original set of 10,000 rules to about 100.

Since there are only 100 rules, robustness is dramatically improved. A fault in a component is now a violation of its own law, not a guess that some designer made beforehand. Generality is also much better. The approach can be applied to more circuits. Its efficiency is good, and its constructibility has become a lot better because now there are only 100 rules to discover. Look at what we've done--we've beaten on the knowledge until there's almost no knowledge in the system about the particular circuit. Instead, we've got a little bit of knowledge, deep knowledge, about the domain of circuits in general, and that deep knowledge applies to all circuits, not just this one.

Next, let's consider the fifth approach to troubleshooting deep knowledge about fault modes. So far we've only talked about a fault as being deviation from expected behavior, but devices also have fault modes, and those fault modes are very important. For example, consider this circuit:



Suppose we measure a voltage at the input A, construct an expectation of what the voltage must be at the output of A using the model for A, and so on through B and E. Now we make a second measurement at the output of E and find it that conflicts. We can immediately deduce that one of A, B, or E is faulted. If there is only one fault in the circuit, D and C are definitely unfaulted. That's as far as we got by the old strategy, but we can actually get a lot further by asking the question: What caused it? Not what caused the fault, but what caused the

expectations to go awry, and the only thing that can cause the expectation to go awry is something wrong with A, B, or E. I'm going to introduce a second propagator. The first propagator propagated behavior, the second one propagates errors or faults. You can only propagate causally back up through an expectation that you've constructed. The only thing that could cause the expectation to be wrong is A, B, or E. So this second propagator inverts expectations that have already been constructed by the first propagator. So, therefore, I need to have more rules about components and it gets rather complicated. In fact, there's quite a few rules one needs in order to run the propagator backwards, but I won't talk about these new rules in this talk.

This reduces the rule set from 100 to 25. Now why does it do that? We did not build this fault propagator for this purpose. We thought it would just give us better explanations, but it actually helped. In the old scheme, a component was either faulted or unfaulted. That means that the only information recorded was whether a component was faulted or not. But suppose I measure a voltage to be low, that might tell me that R6 can't be high. You know that, I know that, but the program with the candidate sets can't represent that fact. So if it makes a second measurement whose entailment is that R6 can't be low, all of a sudden you and I know that if R6 isn't high, and it isn't low, then it's OK. But the candidate set representation won't support that inference. By extending the representation to include fault modes, this information that the fault propagator collects is utilized. In the old scheme, there were many extra rules to handle situations like this. In the R6 example, the R6 rule could be deleted.

The game is--we want to get rid of knowledge. Knowledge isn't power. Knowledge is evil. So we're left with 25 rules for this device. And the first 16 are shown in Figure 4. In doing research, we spent a lot of time staring at these rules to identify what kinds of inferences lay behind them. What really stuck out from looking at those rules is that each depended on a very sophisticated inference on a causal model. So that pushed us into the research we're doing right now: How to use a causal model to troubleshoot a circuit.

Let me briefly illustrate this sixth and final approach to troubleshooting by showing an inference that the other propagation-based troubleshooters can't make but a causal model can. Let's look back at Figure 3. The rule is, "If Q5 is off, then this reference can't be too low." The only way the reference is connected to the rest of the power supply is through Q5. This is a topological inference. That doesn't tell you the rule, it's just an observation to help you understand the rule.

This rule is based on the fact that Q5 disconnects the voltage reference from the power supply. Suppose we have a causal model which identifies the two main control feedback paths of the IP-28. Thus, the way the supply works, is to compute the minimum of the two settings, and that's the output. This minimum is computed through a feedback path in which Q5 is a part. If this voltage reference were too low, it would turn on Q5 causing a symptom at the output. So there's no way that a low voltage on the voltage reference could be contributing to the power supply's faulty behavior if Q5 were disconnected. If Q5 were disconnected (i.e., Q5 off) the output could be high due to the reference being too high, but the output could not be too low due to the reference being too low. That's an example

```

Demon # 1 :
(AND (STATE Q2 SATURATED-OFF))-->>
(OK (BLK CURRENT/SOURCE)
(BLK RECTIFIER/FILTER))

Demon # 2 :
(AND (VOLTAGE (N11 GROUND)
VL-OVER))-->>
(AND (LOCAL!/L REG OVER-V-LIM)
(ASSERT (STATE IP-28 HIGH)))

Demon # 3 :
(AND (VOLTAGE (N11 GROUND)
VL))-->>
(LOCAL!/L REG OVER-I-LIM OK)

Demon # 4 :
(AND (CURRENT N/OP CL-OVER))-->>
(AND (LOCAL!/L REG OVER-I-LIM)
(ASSERT (STATE IP-28 HIGH)))

Demon # 5 :
(AND (VOLTAGE (N11 GROUND)
VL-UNDER)
(CURRENT N/OP CL-UNDER))-->>
(AND (LOCAL!/L IP-28 LOW)
(ASSERT (STATE IP-28 LOW)))

Demon # 6 :
(AND (VOLTAGE (N11 GROUND)
VERY-LOW)
(CURRENT N/OP CL-UNDER))-->>
(AND (LOCAL!/L IP-28 VERY-LOW)
(ASSERT (STATE IP-28 LOW)))

Demon # 7 :
(AND (CURRENT N/OP CL-UNDER))-->>
(ELIM!/L REG OVER-I-LIM)

Demon # 8 :
(AND (CURRENT N/OP CL))-->>
(LOCAL!/L REG OVER-V-LIM OK)

Demon # 9 :
(AND (STATE Q5 OFF))-->>
(ELIM!/L V/DIF/CONT LIMIT-TOO-LOW ALWAYS-ON)

Demon # 10 :
(AND (VOLTAGE (N11 GROUND)
VL-UNDER))-->>
(ELIM!/L REG OVER-V-LIM)

Demon # 11 :
(AND (VOLTAGE (N11 GROUND)
VERY-LOW))-->>
(ELIM!/L REG OVER-V-LIM)

Demon # 12 :
(AND (STATE Q6 OFF))-->>
(AND (ELIM!/L I/VOL/CONT LIMIT-TOO-LOW ALWAYS-ON)
(OK C5))

Demon # 13 :
(AND (STATE Q5 ON))-->>
(LOCAL-OR-FAULTED!/L (I/VOL/CONT ALWAYS-OFF LIMIT-TOO-HIGH)
(V/DIF/CONT ALWAYS-ON LIMIT-TOO-LOW
LIMIT-TOO-HIGH))

Demon # 14 :
(AND (CURRENT C/Q2 LOW))-->>
(ELIM!/L CCS OK)

Demon # 15 :
(AND (VOLTAGE (N4 GROUND)
NORMAL))-->>
(ELIM!/L CCS HIGH-IMPEDENCE)

Demon # 16 :
(AND (CURRENT C/Q2 NORMAL))-->>

```

Figure 4. Demon Rules

of an inference that cannot be captured in the other five troubleshooting approaches. The only reasonable way I've seen to do it is to construct a mechanistic model.

Here's my current game: You give me a schematic for a power supply, the program reads it in, parses it, constructs a causal model, then generates the 25 rules from the causal models. Then I have a complete troubleshooter. I have a troubleshooter that works if you give it a power supply, and it will troubleshoot it. You don't have to sit there and add more things to it. It works solely from the schematic like a human technician.

So look what I've done. I've used theory and inference to beat knowledge to almost nothing. The little bit of knowledge that I do have is about circuits in general. I have achieved robustness, generality, efficiency, and constructibility. But, there's a very high price: Why does it work and what did I have to do? It works because there was a deep theory of circuits, and it had to be made computational. The fundamental business of AI is getting deep theories of various domains and making them computational. But don't send away for my troubleshooter. I haven't got it completely working yet and there are still a few problems.

Thank you.