Technical Report 1074

Model-Based Troubleshooting of Digital Systems

Walter Charles Hamscher

MIT Artificial Intelligence Laboratory

Model-Based Troubleshooting of Digital Systems

by

Walter Charles Hamscher

Revised version of a thesis submitted to
the Department of Electrical Engineering and Computer Science
on 12 August 1988 in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science

Abstract

This thesis describes a methodology, a representation, and an implemented program for troubleshooting digital circuit boards at roughly the level of expertise one might expect in a human novice. Existing methods for model-based troubleshooting have not scaled up to deal with complex circuits, in part because traditional circuit models do not explicitly represent aspects of the device that troubleshooters would consider important. For complex devices the model of the target device should be constructed with the goal of troubleshooting explicitly in mind. Given that methodology, the principal contributions of the thesis are ways of representing complex circuits to help make troubleshooting feasible. Temporally coarse behavior descriptions are a particularly powerful simplification. Instantiating this idea for the circuit domain produces a vocabulary for describing digital signals. The vocabulary has a level of temporal detail sufficient to make useful predictions about the response of the circuit while it remains coarse enough to make those predictions computationally tractable. Other contributions are principles for using these representations. Although not embodied in a program, these principles are sufficiently concrete that models can be constructed manually from existing circuit descriptions such as schematics, part specifications, and state diagrams. One such principle is that if there are components with particularly likely failure modes or failure modes in which their behavior is drastically simplified, this knowledge should be incorporated into the model. Further contributions include the solution of technical problems resulting from the use of explicit temporal representations and design descriptions with tangled hierarchies.

Acknowledgements

Success is humbling: I am permanently indebted to the many fine people who have done so much for me during my years at MIT.

Randall Davis, my advisor, spent countless hours providing intellectual guidance, unflagging moral and financial support, and of course the all-important comedic perspective.

My readers Ramesh Patil, Howard Shrobe, and Peter Szolovits, spent a great deal of time with me and with this document, providing their own indispensably unique viewpoints.

All the usual suspects at the weekly Hardware Troubleshooting group meetings, including my friends Meyer Billmers, Choon Goh, Hal Haig, Paul Resnick, Mark Shirley, Reid Simmons, Raúl Valdés-Pérez, Jeff Van Baalen, Dan Weld, and Peng Wu, provided close comradeship and a constant stream of intellectual stimulation and feedback on my work and on my presentations of it. Brian Williams deserves special mention for providing so many patient explanations, penetrating observations, and encouraging words.

Comments from Eugene Charniak, Johan de Kleer, Tom Knight, Drew McDermott, John McDermott, Chuck Rich, Gordon Robinson, Bill Swartout, Dick Waters, and Mike Wellman, sometimes only a single incisive sentence, all helped to clarify my thinking at crucial moments. Everyone at the MIT Artificial Intelligence Laboratory contributed to its atmosphere of challenge and excitement.

The Joshua group and others at Symbolics, including my friends Steve Anthony, John Aspinall, Brian Bauer, Bob Cassels, Jackie Covo, Doug Evans, John Hotchkiss, Jim Loftus, Neil Mayle, and Steve Rowley, taught me much about AI in the world outside this ivory tower.

The love and encouragement of my parents and of my wife, Cristina Ciro, enabled me to carry the work through to its completion. Their patient and unwavering support is most humbling of all. This is their success, too!

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the author's artificial intelligence research on troubleshooting is provided by the Digital Equipment Corporation, Wang Laboratories, Symbolics, and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

To my Family

Contents

1 Introduction **1**
- 1.1 Model-Based Troubleshooting 1
- 1.2 A Troubleshooting Scenario 6
- 1.3 Contributions 11
- 1.4 Organization 14

2 Background **16**
- 2.1 The Symptom-Based Approach 16
  - 2.1.1 Dealing with Uncertainty 17
  - 2.1.2 Organizing Knowledge 18
  - 2.1.3 Diagnosing Multiple Diseases 18
  - 2.1.4 Summary of Symptom-Based Approaches 19
- 2.2 The Model-Based Approach 19
  - 2.2.1 Modeling 20
  - 2.2.2 Behavior Prediction 23
  - 2.2.3 Candidate Generation 24
  - 2.2.4 Discrimination 31
  - 2.2.5 Hierarchic Diagnosis 33
  - 2.2.6 Summary of the Model-Based Approach 35

3 Troubleshooting Scenarios **38**
- 3.1 Clock Generator Examples 40
  - 3.1.1 Troubleshooting the Clock Generator 41
  - 3.1.2 Morals of the Clock Generator Example 42
- 3.2 Audio Decoder Examples 43
  - 3.2.1 Functional Organization of the Audio Decoder 44
  - 3.2.2 Physical Organization of the Audio Decoder 49

3.2.3 Audio Decoder Example I 49
3.2.4 Audio Decoder Example II 52
3.2.5 Audio Decoder Example III 53
3.2.6 Audio Decoder Example IV 54
3.2.7 Summary of the Audio Decoder Examples 57
3.3 Input Encoder Examples 57
3.3.1 Functional Organization of the Input Encoder 57
3.3.2 Physical Organization of the Input Encoder 59
3.3.3 Expected Behavior of the Input Encoder 62
3.3.4 Finding a faulty Input Processor 63
3.3.5 Finding a faulty Console Controller 67
3.4 Summary of Troubleshooting Scenarios 69

4 Representing Circuit Structure 70
4.1 Physical Organization 71
4.1.1 Primitive Components 71
4.1.2 BASIL 74
4.1.3 The Physical Part-Of Hierarchy 76
4.2 Functional Organization 78
4.2.1 The Functional Part-Of Hierarchy 80
4.2.2 Principles for Structural Composition 82

5 Representing Circuit Behavior 86
5.1 TINT 90
5.1.1 Signals 90
5.1.2 Rules 92
5.1.3 Signal Histories 93
5.1.4 Equality 95
5.1.5 Summary 96
5.2 Combinational Behaviors 98
5.3 Sequential Behaviors 103
5.4 Abstractions 107
5.4.1 Temporal Abstractions 110
5.4.2 Composite Abstractions 119
5.4.3 Summary of Abstractions 122
5.5 Event Preservation 124
5.6 Reduction 131

5.7 Synchronization 134
5.8 Encapsulation 140
5.8.1 The Reset Hold Counter 140
5.8.2 The Audio Counter 145
5.8.3 Microprocessors 147
5.8.4 Abstract Buffers 148
5.8.5 Programmed Microprocessors 149
5.9 Related Work 163
5.9.1 Temporally Quantified Statements 163
5.9.2 Intervals and Constraints on Intervals 163
5.9.3 Persistence 164
5.9.4 Temporal Indexing 164
5.10 Summary of Behavior Representation 165

6 Representing Faults and Misbehaviors 166
6.1 Failure Likelihoods 168
6.2 Representing Syndromes 171
6.3 Principles for Using Syndromes 175
6.4 Consequences of Using Syndromes 180
6.5 Summary of Faults and Misbehaviors 183

7 Troubleshooting 184
7.1 Conflicts and Candidates 185
7.2 Decomposition 187
7.3 Ranking and Refinement 193
7.4 Making Observations 200
7.4.1 Prediction Strength and Probe Selection 202
7.4.2 Temporal Quantification and Granularity 204
7.5 Evaluation 206
7.5.1 Coverage 206
7.5.2 Resolution 207
7.5.3 Speed 209
7.6 Summary 210

8 Conclusions and Future Work 211
8.1 Engineering Issues 214
8.2 Deriving the Representation 216

8.3 Generalizing the Methodology 217

A Scenario Transcripts **220**

A.1 Clock Generator Example 221

A.2 Audio Decoder Example I 223

A.3 Audio Decoder Example I with Syndromes 225

A.4 Audio Decoder Example II 228

A.5 Audio Decoder Example II with Syndromes 232

A.6 Audio Decoder Example III 235

A.7 Audio Decoder Example III with Syndromes 241

A.8 Audio Decoder Example IV 247

A.9 Audio Decoder Example IV with Syndromes 252

A.10 Input Encoder Example I 258

A.11 Input Encoder Example II 263

B Abstractions and Behaviors **268**

C Reset Hold Counter Behavior **273**

D Audio Counter Behavior **284**

E The Switch Level Model **288**

E.1 Pins and other Connections 288

E.2 Resistors 290

E.3 Switches 291

Bibliography **306**

List of Figures

1.1 Model-Based Troubleshooting 2
1.2 Model-Based Troubleshooting Problems 3
1.3 A Portion of the Console Controller Board 7
1.4 Likely Suspects After Probing *Interrupt* 9
1.5 Likely Suspects After Probing *Reset* 10
1.6 Likely Suspects After Probing *Constant* 10

2.1 Behavior Prediction Example 23
2.2 Reasoning from Effects to Causes 24
2.3 Behavior Prediction Example 27
2.4 Reasoning from Effects to Causes 27
2.5 Discrepancies Produce Conflicts 28
2.6 If x is not 1, Only A Could be Broken 29
2.7 Inverter B Could be Pulling x Down 30
2.8 A Short Could be Pulling x Down 30
2.9 Diagnosis of Adder-1 34
2.10 Diagnosis of Adder Substructures 36

3.1 Overall Troubleshooting Program Organization 39
3.2 Clock Generator Schematic 41
3.3 Clock Generator Structure 41
3.4 Audio Decoder Schematic 44
3.5 Audio Decoder Functional Organization 46
3.6 Signal with Too Many Zero Crossings in its First Derivative . 47
3.7 Audio Decoder Physical Organization 50
3.8 Signal with Too Many Zero Crossings 55
3.9 Internal Structure of CSB01 56
3.10 Input Encoder Functional Organization 58

3.11 Input Encoder Schematic 60
3.12 Input Encoder Physical Organization 61

4.1 Chip Cross Section 72
4.2 Abbreviated AKO hierarchy 74
4.3 A Portion of the ppart-of Relation 77
4.4 Expanded AKO Hierarchy 79
4.5 JK Flipflop Unencapsulated 80
4.6 JK Flipflop Encapsulated as a Toggle 81
4.7 Power Ports of Chip U30 and its Chiplets 83
4.8 Encapsulating Reconvergence 84
4.9 Encapsulating a Sequential Loop 84

5.1 TINT Signal History Example 95
5.2 Combinational Behavior Example 101
5.3 Example with Register Behavior Rules 106
5.4 Abstractions and Behaviors 107
5.5 Example of Abstractions and Behaviors 108
5.6 Sufficiently Complex Abstractions Make Any Behavior Trivial 109
5.7 The Behavior of a Counter with Respect to a "Counting" Abstraction 109
5.8 Frequency Divider Implemented with JKFFs 129
5.9 Register Abstractions 134
5.10 Shift Register as Cascade 136
5.11 Reset Hold Counter 141
5.12 Reset Hold Counter Three State Automaton 142
5.13 Audio Counter 145
5.14 The Third Key is Pressed 150
5.15 Functional Organization of Console Controller 156
5.16 Components of B 157
5.17 Components U and C Together form Component E 159
5.18 State Diagram of E 159

6.1 Clock Generator 170
6.2 Audio Counter 177
6.3 Every Component is a Candidate 181

7.1 Predications, Assumptions, and Environments 185

7.2 Physical Organization of Four-Bit Adder 189
7.3 Functional Organization of Four-Bit Adder 189
7.4 Physical and Functional Organizations 190
7.5 Physical and Functional Decompositions of the Four-Bit Adder 191
7.6 XPART-OF Relations in the Four-Bit Adder 192
7.7 Clock Generator 194
7.8 Distinguishing Between Diagnoses 208
7.9 Detail of Audio Decoder 209

C.1 Reset Hold Counter 274

E.1 Typical Switch-Resistor Combination 293
E.2 TTL Inverter as Modeled with TINT 294
E.3 nMOS Inverter as Modeled with TINT 295
E.4 Impasse Example 296

Chapter 1

Introduction

A field engineer plugs in a broken circuit board, makes a half dozen simple probes with an oscilloscope, and after ten minutes ends up swapping a chip, which fixes the problem.

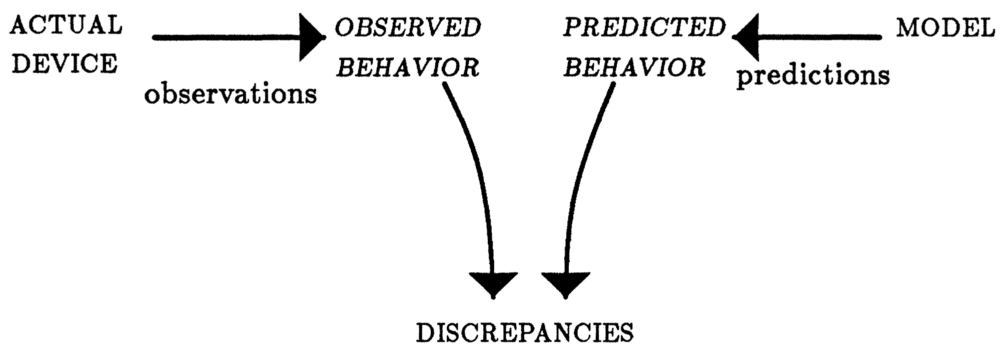
A model-based troubleshooting program spends a day simulating the expected behavior of the same misbehaving board, and requests that a logic analyzer be used to capture a certain subset of the signals. After some hours of computation it concludes that any of the 40 chips or 400 wires on the board could be responsible for the misbehavior.

What does the field engineer know that the program does not? How can a model-based troubleshooting program represent and use that knowledge? Both the program and the field engineer have the circuit schematic and the specifications of the individual chips. The field engineer additionally has expectations about the design of the circuit, expectations about which signals in the circuit ought to be changing and how fast, and expectations about the kinds of failures that are likely to occur in digital circuits. Incorporating this knowledge into the circuit model makes it possible to be more discriminating in the generation of diagnoses and more efficient in the use of observations.

1.1 Model-Based Troubleshooting

Model-based troubleshooting is driven by the interaction of observation and predictions (Figure 1.1). A device model produces predictions about what ought to be observed; comparison with observations of the actual device produce discrepancies; these discrepancies are then traced to their possible underlying causes in the model and repairs of the actual device proposed.

Figure 1.1: Model-Based Troubleshooting

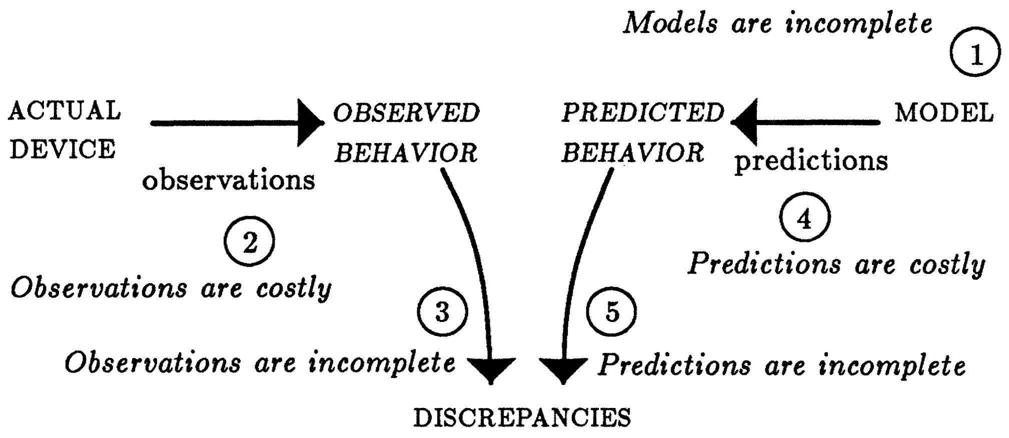


This report describes a model-based troubleshooting program. Its primary input is a model of a digital circuit that is a network of components and connections. Each component has a description of its dynamic time-dependent behavior and each connection transmits signals between components. The secondary input to the program is a description of the stimuli presented to the circuit and observations of its actual responses. The model uses those stimuli to predict what the outcomes of observations ought to be. When discrepancies are discovered, the program produces lists of components that could be responsible for the discrepancies, ranked by their relative likelihood. The program interactively suggests what observations should be made next to discriminate among these possibilities, then uses any new observations to incrementally focus on the correct diagnosis.

Model-based troubleshooting has been extensively demonstrated on simple devices. One of the prime motivations of this work is to scale up model-based troubleshooting techniques to deal with significantly more complex devices. The fundamental problems in scaling model-based troubleshooting technology to do this can be understood as problems within each element of the paradigm (Figure 1.2). These five problems and their solutions are discussed individually below.

Models are incomplete. No model can possibly capture every detail of

Figure 1.2: Model-Based Troubleshooting Problems



the actual device. Lack of detail in the device representation means that some failures will be indistinguishable and others will be misdiagnosed. For example, if a wire connecting several terminals is represented as a single component, then the program will diagnose a break anywhere along the wire as a failure of the whole wire. If the model says that the only devices affecting the state of the wire are the ones that it was meant to connect, then the troubleshooting program will misdiagnose a short between that wire and another as having been caused by one or more other failures. The selection of the primitive elements of the device representation constitutes a commitment to a set of failures worth identifying and worth distinguishing from each other.

Models are incomplete, but the consequences of that incompleteness can be controlled in part by the choice of primitive elements and their connections to each other. Principles are needed for making these choices in a way that sacrifices completeness in favor of efficiency, since the aspiration is to troubleshoot circuits with many thousands of wires, transistors, and interactions between them. One such principle is that physically separate components with indistinguishable failure effects can be treated as a single component. Another principle is that components whose failures result in the same repair can be treated as a single component. A third principle is that unlikely failures are not worth representing explicitly, so that components whose failures are individually very unlikely can all be treated as a single aggregate component whose failure is more likely. These principles introduce additional approximations into a device model that will make some component failures indistinguishable from one another. A deeper problem arises from the fact that any model explicitly represents only some of the possible interactions between components; the program will misdiagnose any failures involving interactions that the model does not represent. The standard example is an unintentional short between two wires that are unrelated in the circuit structure diagram. The best that the troubleshooting program will do is to diagnose this as two failures, one in each wire. The approach taken in this work is not a general solution: at any given level of detail, decisions about which interactions between components ought to be represented are made solely on the basis of what is needed to explain the normal operation of the device. In the case of wires, only the interactions with the devices they are supposed to be connected to are represented, hence shorts are misdiagnosed. When it comes time to repair the two wires one may assume that their true (mutual) problem will be discovered by visual inspection.

Observations are costly. Taking measurements is nearly always appropriately regarded as being more costly than computation spent on choosing that measurement. The problem that scaling brings is that the more complex the device, the more events there are to observe, and the shorter the intervening intervals, the more difficult they are to observe. It is, for example, more costly to set up a logic analyzer to capture digital signals at particular moments than it is to observe whether they are staying at a constant zero or one.

Observations are costly, and although there is nothing that can be done about this directly, the device model can describe signals in ways that are relatively cheap to observe. For example, it is easier to observe whether a particular signal is rising or falling than to observe its changing value at every moment. This is an example of a useful temporal abstraction; a long sequence of changes of value can be summarized into a simple description that is stable over a longer time interval. A behavior model can use this kind of temporally abstract observation to make other temporally abstract predictions, without requiring that any explicit deductions ever be made about the individual changing values. As a general principle temporal abstractions are useful because they provide a better match to the observations that can be made cheaply.

Observations are incomplete and imprecise. Discrepancies can only be detected where observations can be made. But even when observations can be made, they may be too coarse to detect discrepancies with the model. For example, if the model predicts that a certain current should be flowing in a wire, but the troubleshooter can only measure currents to within 20%, then the current could actually be wrong and yet yield no apparent discrepancy, hence yield no new information. One of the consequences of incomplete observations is that there will inevitably be pairs of diagnoses that cannot be discriminated, since their only difference might be in some unobservable feature. Inability to make certain observations economically imposes limits on the ability of the troubleshooting program to isolate faults.

Because observations are incomplete, ambiguity among the logically possible diagnoses is inevitable. If the troubleshooting goal is to find the most *likely* diagnosis, however, other sources of information are available. One of these sources is information about the relative failure rates of different physical components, from which the troubleshooter can produce a rank ordering of the diagnoses by plausibility. A related source is information about how components usually fail and what misbehaviors they produce; this can be used to refine the likelihood estimates for some diagnoses. These sources of knowledge alleviate the indiscriminacy caused by incomplete observations because they can be used to discount unlikely diagnoses and leave the remaining (relatively more likely) ones behind.

Prediction is costly. It is impractical within a troubleshooting session to simulate an entire circuit board at the gate level for more than a few clock cycles. The culprit is not the structural complexity of the board in number of gates or wires. The culprit is the complexity of the behavior — the number of events that happen and need to be simulated. Waiting for more computing power to apply to the problem is not a solution if the boards to be diagnosed themselves get faster and more complex.

Prediction is costly, but this can be addressed by using temporally abstract behavior descriptions. Temporal abstractions can summarize many individual events into an aggregate description stable over a longer interval. For example, a given signal may be described as a sequence of many thousands of individual alternating zeroes and ones, or more abstractly in terms of the number of falling edges that have appeared, or even more abstractly as the number of one-to-zero cycles per unit time. Although the value of the underlying signal may be changing many times per second, the average number of cycles per unit time may be relatively stable. Descriptions that are stable in this way are less costly to make predictions from. For example, the troubleshooting scenario to be presented shortly is simple because the behavioral complexity of microprocessors can be reduced to a simple relationship between the rates of change at their inputs and outputs.

Predictions are incomplete. A consequence of using abstract models of behavior to achieve more economical prediction is that the resulting predictions may be imprecise or ambiguous. Predictions that are too coarse make it difficult to detect discrepancies with observations, and this in turn sacrifices some of the ability of the program to isolate faults.

Economical predictions are incomplete, but the indiscriminacy that results can be alleviated by using multiple levels of behavioral abstraction. If needed, more detailed predictions can be made for only a subset of the entire device. This may allow more discrepancies to be detected and thereby rule out some diagnoses.

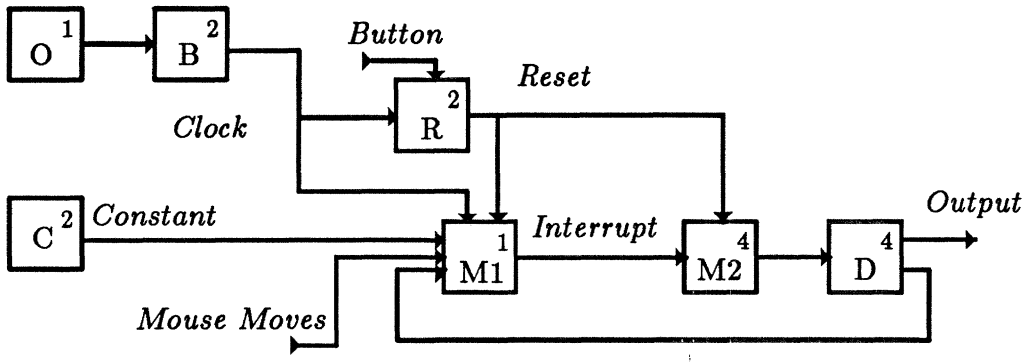
1.2 A Troubleshooting Scenario

The troubleshooting program described in this report uses a rich and multilayered circuit model that is designed to address the problems identified above. The model represents the physical organization in terms of chips, wires, and so forth, and represents the functional organization in terms of how its parts interact to achieve the overall intended behavior. Its levels of detail range from a qualitative model of resistors and switches up to arbitrarily large computational modules. It represents the behaviors of components using both traditional digital abstractions and a novel set of temporal abstractions that describe signals in terms such as *cycles*, *frequency*, and *change*. Finally, it incorporates knowledge not just about how the circuit components should work, but for a few, how they break and how often. Only one circuit has been modeled this way, but it is large, complex, internally diverse, and real: a portion of the Symbolics 3600 Console Controller Board that contains two microprocessors (both running programs with several hundred instructions), thirty supporting chips, and one hundred sixty wires.

Seven troubleshooting scenarios using this circuit will be presented in this document. One of these scenarios, presented here in abbreviated form, serves to illustrate the distinctive features of the circuit model and the interaction of the troubleshooting program with it.

The Console Controller Board is responsible for transmitting keystrokes and mouse motions to the host computer and for decoding the video signal coming from the host for display on a CRT and the audio signal for output to a speaker. Some keystroke sequences can change the volume of the speaker, the brightness of the CRT, and so forth. Figure 1.3 shows abstractly a few of the components (boxes) and the signals through which they interact (arrows).

Figure 1.3: A Portion of the Console Controller Board



Each small superscript represents the number of chips in that component; there are 16 in all. The oscillator O produces a clock signal that is buffered by B and sent on to two places: the reset circuitry R and to a microprocessor M1. The microprocessor M1 polls the mouse inputs. Each tenth of an inch of mouse motion along its x or y axes causes M1 to interrupt a second microprocessor M2 with a two-byte message. M2 responds to the interrupt through some bus control circuitry D. After receiving the two-byte message M2 then sends the message on to the host, again through the bus control circuitry D. The host displays the changed mouse position on the screen.

Suppose the Console Controller Board reset button is pressed and the mouse rolled around for a couple of seconds. The model predicts that if all 16 chips are working, then mouse motion will be observed at *Output*. The model is too coarse to predict how fast or how far the cursor will move on the screen — it predicts only that motion will be observed. This temporally abstract behavior is both more efficient to make predictions from and easier to observe than the traditional clock-cycle-by-clock-cycle model of digital circuit behavior.

But suppose the mouse cursor does not move at all. The program indicates that any one of the 16 chips might be broken; each chip is a suspect. There are now many possible signals to probe, and the program ranks them. The likeliest chip to fail by far is the onboard oscillator O. The program suggests probing its output; suppose it is observed to have a frequency of approximately 10 Mhz.

The oscillator O can be discounted as an unlikely suspect using knowledge in the model about how some components fail. The model says that when oscillators fail, they usually fail catastrophically, producing an output frequency of 0. Because the signal was observed to be changing, the program concludes that the oscillator chip is probably not responsible. It is still a suspect, just a relatively unlikely one. This leaves 15 chips as likely suspects.

The program now needs to suggest another probe. To suggest a probe it considers the predictions that the model makes at each signal. For example, the model predicted that the output of the oscillator O should have frequency 10 Mhz, and the probe verified this. The model also predicts that the *Clock* signal should have frequency 5 Mhz. The representation of these clock signals in terms of their frequencies is an example of a temporal abstraction; millions of underlying events (rising and falling edges) have been abstracted into a simple description that is easy to reason about and easy to observe.

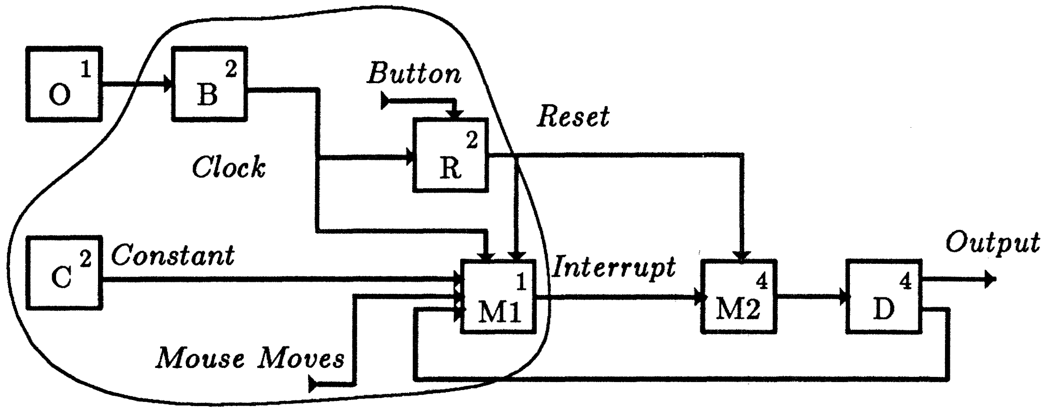
Although the model represents many signals in temporally abstract ways, there are other signals for which the standard digital vocabulary suffices. For example, the *Constant* output of C is a constant 1 throughout the entire session, and the model predicts that. Also, the *Reset* signal should be asserted while the reset button is pressed and unasserted otherwise, and the model predicts that as well.

These predictions — that the clock frequency is 5 Mhz, and so forth — can be used in subsequent predictions. The temporally abstract behavior model for the first microprocessor M1 says that if the *Clock* input is 5 Mhz, the *Constant* input is 1, and the *Reset* signal is not asserted, then the microprocessor is running. While M1 is running, each movement of the mouse results in the *Interrupt* line being asserted. If all that is known is that the mouse is moving around, the model does not predict exactly when it will be asserted; rather it predicts that the signal will be *changing* while the mouse is moving and a constant 1 value otherwise.

The model makes many other predictions, but these are all that will be needed in this example. The important one at the moment is the prediction that *Interrupt* signal will be changing while the mouse is moving. This prediction depends on eight chips working properly, those in all components except M2 and D.

The probe that the program now suggests is the *Interrupt* output of M1. Suppose the interrupt line is probed, revealing that it is a constant 1 even while the mouse is rolled around. This is a discrepancy, since it was supposed to be changing so long as those eight chips were working properly. One of the chips was the oscillator, which has been shown to be an unlikely suspect; this leaves seven as likely suspects (Figure 1.4).

Figure 1.4: Likely Suspects After Probing *Interrupt*



The model predicted that the *Reset* signal should be asserted just while the reset button was pressed, so long as the five chips in O, B and R were working. Probing the *Reset* signal reveals that upon pressing the button it is asserted, then unasserted. This means that the chips in R are no longer suspects, since their failure could not explain the observations made. Now there are 5 likely suspects (Figure 1.5).

The model predicted that the *Constant* signal should be 1 throughout the session, so long as the chips in C were working. Probing this signal reveals that it is indeed 1, so the chips in C are no longer suspects. Now there are 3 likely suspects (Figure 1.6).

Finally, a probe of the *Clock* signal reveals that it has frequency around 5 Mhz. The model says that if the clock input to M1 has a high enough

Figure 1.5: Likely Suspects After Probing *Reset*

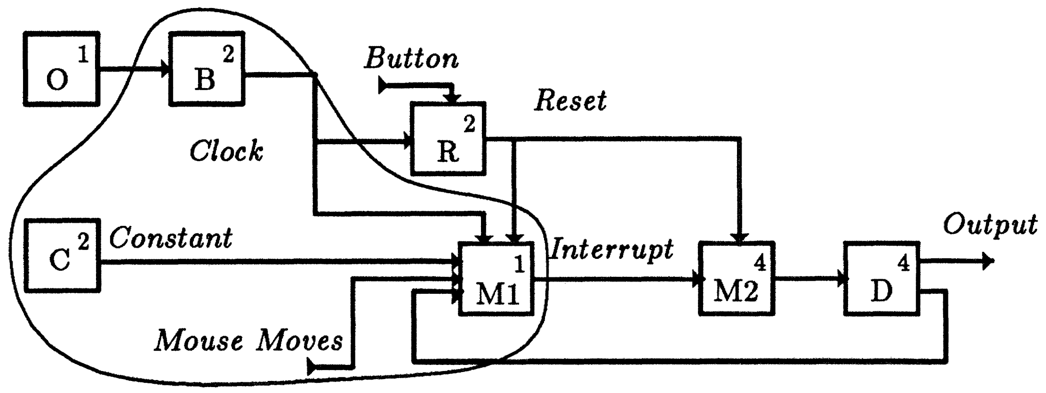
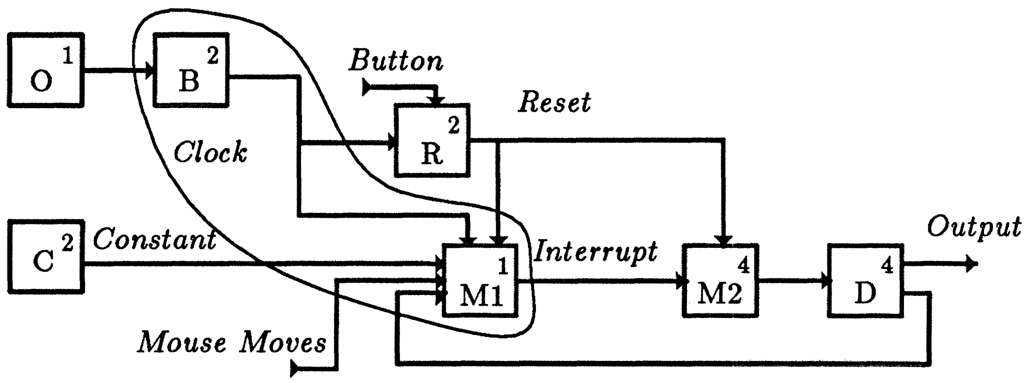


Figure 1.6: Likely Suspects After Probing *Constant*



frequency and the reset input is not asserted, then the microprocessor should be running. This means that the *Interrupt* signal should be changing, which contradicts previous observations. Hence M1 is the only remaining suspect and the program terminates.

The interesting thing about this scenario is that it is so simple compared to the underlying complexity of the real circuit. The circuit is structurally complex; there are thousands of transistors in the chips, hundreds of possible flaws in the wires alone. It is behaviorally complex; consider all the microprocessor instruction cycles that occurred during the one second of mouse motion. People can troubleshoot the circuit without thinking about all those details, and the program can troubleshoot it without explicitly representing them.

The important thing about the model is not that it uses abstractions to deal with complexity; any representation does that. The important idea is that there are structural and behavioral abstractions appropriate to troubleshooting. Temporal abstractions, in particular, allow the program to avoid simulating long sequences of events and instead reason in terms of "moving" mice, "running" clocks, "changing" signals, and so forth. There are also principles by which those abstractions can be manually applied to a complex circuit to construct the rich representation that makes troubleshooting of complex devices tractable. The model of the Console Controller Board is appropriate for model-based troubleshooting because it was constructed according to those principles.

1.3 Contributions

This thesis presents a methodology, a representation, and an implemented program for troubleshooting digital circuit boards at roughly the level of expertise of a human novice.

The methodological claim is that existing methods for model-based troubleshooting have not scaled up to deal with complex digital circuits because traditional circuit models do not explicitly represent aspects of the device that troubleshooters would consider important. For complex devices the model of the target device should be constructed with the goal of troubleshooting explicitly in mind.

Given that methodology, there are principles by which complex circuits can be represented so as to make those important aspects explicit and thereby help make the troubleshooting task tractable. Some of the salient principles follow.

One set of principles concerns how the structure of a given circuit should be represented.

- Components in the representation of the physical organization of the circuit should correspond to the possible repairs of the actual device.

The representation of physical organization plays a central role in the troubleshooting program, and the program represents all of its diagnoses in terms of the physical components that could be damaged. In the scenario presented earlier, for example, the diagnoses were expressed in terms of chips, which are "repaired" by replacement. Making the elements of this representation correspond to possible repair actions ensures that the troubleshooting program will not waste effort trying to discriminate between diagnoses that have identical repairs.

- Components in the representation of the functional organization of the circuit should facilitate behavioral abstraction.

The only role that an explicit representation of functional organization plays in model-based troubleshooting is to make behavior prediction more efficient. For example, the only reason that the component M2 exists in the model is because the combined behavior of the four chips inside it can be described more simply in the aggregate than individually. In extracting the functional organization from a raw schematic the modeler need only represent what will make the behavior easiest to reason with, rather than necessarily what the designer had in mind.

A second set of principles concerns the representation of circuit behavior.

- The behavior of components should be represented in terms of features that are easy for the troubleshooter to observe.

Some features of time-varying signals are easier to observe than others. The frequency of a clock, for example, is easier to observe than the timing of each of its individual transitions. Expressing the behavior of components in the terms that are more easily observed is a way of choosing where to sacrifice precision in favor of efficiency.

- The behavior of a component for which changes on its inputs always results in changes on its outputs should be represented in temporally coarse terms.

A powerful representation technique uses relationships between component inputs and outputs in terms that are stable over long periods of time or that summarize much activity into a single parameter. In the troubleshooting scenario, the number of mouse step increments over a period of seconds (a single parameter describing much activity) determined the number of times the interrupt line would be asserted over that period. Such relationships can be derived when each individual change results in one or more other changes.

- A temporally coarse behavior description that only covers part of the behavior of a component is better than not covering any at all.

Although the full behavior of a component may be too complex to reduce to a simple relationship between (say) the number of changes on its inputs and the number of changes on its outputs, there may be such a relationship that involves only a subset of its inputs, assuming that the others are held constant. In the case of the microprocessor, for example, the relationship between the mouse motion inputs and interrupt output holds only so long as the clock input is running and the reset input is not asserted. Since the troubleshooting program will eventually use the more detailed behaviors as long as the diagnosis remains ambiguous, no diagnostic resolution will be lost by only representing a subset of the possible behaviors abstractly.

- A sequential circuit should be encapsulated into a single component to enable the description of its behavior in a temporally coarse way.

Although the individual behaviors of the components in a sequential circuit may not lend themselves to temporally coarse descriptions, the loop may be performing a simple function when taken as a whole. For example, the R component in the troubleshooting scenario is actually a sequential circuit with 2^{14} distinct states. When viewed in temporally coarse terms, however, there is a simple correspondence between the states of the button and the state of the output. Encapsulating the group of components makes it possible to reason about its behavior in a temporally coarse way, and as in the troubleshooting scenario described, it may not be necessary to ever consider the details of its behavior.

A final set of principles concerns what knowledge about failures should be represented explicitly.

- An explicit representation of a given component failure mode should be used if the underlying failure has high likelihood.

Components break in the field in certain ways much more often than other ways. Chips, for example, fail more often with breaks in the tiny wires that connect their pins to the silicon chip inside than in other ways. The benefit of knowledge about such failures comes when they are inconsistent with the symptoms, since this can reduce the ambiguity among the possible diagnoses.

- An explicit representation of a given component failure mode should be used if the resulting misbehavior is drastically simpler than the normal behavior of the component.

If a component with normally complex behavior has some internal fault or faults that cause it to misbehave catastrophically, then any partially correct behavior observed for the component makes it a less likely suspect. In the troubleshooting example, the oscillator was known to fail in a way that made it produce a zero output frequency, and that misbehavior was easy to rule out even though the measurement of its output was imprecise. The benefit of knowledge about these failure modes is especially great when the misbehavior has high likelihood as well.

The implemented model of the Console Controller Board is a concrete embodiment of the methodology and representation principles. The troubleshooting program that uses that model is an extension of standard model-based troubleshooting technology, incorporating solutions to technical problems of (i) hierarchic diagnosis with multiple and tangled hierarchies (ii) integration of explicit knowledge about failure modes into a framework for diagnosing multiple faults, and (iii) troubleshooting circuits with time-dependent behavior.

1.4 Organization

This document is primarily organized by the different kinds of circuit knowledge to be represented. Preliminary background material is contained in Chapter 2, which presents an overview of knowledge-based automated diagnosis, especially model-based troubleshooting. Chapter 3 presents the troubleshooting scenarios for the Console Controller Board so as to provide context for the many details to follow. The next four chapters contain the essential ideas. Chapter 4 presents a representation for circuit structure motivated by troubleshooting requirements. Chapter 5 contains the bulk of the document and describes a representation for circuit behavior using multiple temporal abstractions and a temporal reasoning program for predicting behavior using those same abstractions. Chapter 6 describes how faults and misbehaviors are modeled and how this knowledge is used by the troubleshooting program to heuristically discount unlikely diagnoses. Chapter 7 presents the details of the troubleshooting engine and how it interacts with the choices made in representing circuit structure and behavior. Finally, Chapter 8 summarizes and presents ideas for future work. Sections on related research are distributed throughout the individual chapters.

Chapter 2

Background

A number of knowledge-based programs for automated diagnosis have been built for a variety of domains using a variety of implementation technologies. These programs can be characterized by the knowledge that they represent explicitly: (i) associations between underlying diseases or faults and their consequences for the system as a whole, as opposed to (ii) knowledge about the parts of the system and how they interact to produce its overall behavior. In medical diagnosis, for example, the contrast is between knowledge about diseases and their symptoms versus knowledge about the underlying mechanism; it is the difference between knowledge that emphysema causes shortness of breath versus knowledge that CO_2 exchange is proportional to the surface area of the alveoli. Programs that rely on the former type of knowledge will be termed *symptom-based* and the latter *model-based.* A number of programs incorporate both kinds of knowledge, but for any given program it is typically clear which one predominates. A brief review of each paradigm is presented below. One particular program for model-based diagnosis will be presented in some detail, since it provides the basis for the troubleshooting technology in this report.

2.1 The Symptom-Based Approach

One approach to automated diagnosis is to organize the program as a database that associates underlying diseases (faults) with their outward symptoms (manifestations). To find the underlying problem from a set of symptoms requires straightforward lookup or pattern matching. The notion of a "fault dictionary" is the canonical example of this approach. The principal difficulty in this approach revolves around the coverage of diseases in the knowledge base. First, associations between single diseases and their symptoms does not easily support reasoning about interactions between diseases. Second, even if multiple simultaneous diseases can be handled the program is limited to considering those individual diseases that were anticipated and explicitly included by the knowledge base builder — there is no theory about how to enumerate the possible diseases of a given system. Third, given a knowledge base intended to be used for diagnosing a particular system there is no principled way to modify the knowledge base when there has been a change in the design (or in our understanding) of that system. Although the paradigm has these inherent limitations and is not used here, some important techniques that generalize beyond it were first developed within this tradition: techniques for dealing with uncertainty, for organizing large knowledge bases, and for dealing with multiple diseases. These techniques are each treated briefly below.

2.1.1 Dealing with Uncertainty

The notion of a disease-symptom database requires some elaboration in domains for which the underlying diseases have widely varying likelihoods and for which the associations between diseases and symptoms is less than certain. One approach is to assign prior probabilities to the diseases, assign conditional probabilities to the symptoms given each disease, and use Bayes' Theorem to find the likeliest disease given a set of symptoms [Szolovits78]. Many automated diagnosis systems use statistical information in this form in spite of the large number of conditional probabilities needed when diseases or symptoms are not independent. One reason for the enduring popularity of the probabilistic framework is that it allows the use of decision theoretic techniques to choose observations that are most likely to reduce the ambiguity among competing diagnoses. Estimating ambiguity using Shannon entropy and choosing the next observation based on a one-ply lookahead turns out to provide good results on average [Gorry73]. A non-Bayesian approach to dealing with uncertain knowledge is taken by the MYCIN program [Shortliffe76], which computes "certainty factors" for its conclusions, but it suffers from the same difficulties with interacting diseases as Bayesian approaches.

2.1.2 Organizing Knowledge

Obtaining diagnostic coverage of any interesting domain requires the maintenance of a large knowledge base. This in turn implies the need for principles for organizing this knowledge. Organizing knowledge about diseases, symptoms, and diagnostic procedures into frames [Minsky75] appears in the diagnosis program PIP [Pauker76]. The use of frames implies no commitment as to whether knowledge about diseases, symptoms, or causal mechanisms will be stored; rather it allows modularization of the knowledge base and thereby simplifies its maintenance. The organization of diseases and their symptoms into specialization hierarchies, as in the internal medicine diagnosis program INTERNIST [Pople82], is an elaboration of this idea. A hierarchic organization makes only a minimal commitment to the character of the knowledge, but it does allow the program to deal with groups of related diseases more efficiently. A stronger organizing principle appears in the glaucoma diagnosis program CASNET [Kulikowski82], in which knowledge is organized around disease states and their temporal progression. This network of states and their successor relationships was intended to represent a causal explanation of the disease. Although the use of this knowledge in CASNET is probabilistic and not substantially different from other symptom-based programs, it was recognized that causality could be a powerful organizing principle because the knowledge acquired from domain experts is often couched as categorical explanation that can be translated into causal terms.

2.1.3 Diagnosing Multiple Diseases

Among the most difficult cases in medicine and other diagnostic tasks are those in which more than one underlying disease or fault is present. One approach is to assume that all underlying diseases are statistically and causally independent. The program can then simply evaluate the likelihood of every disease individually. This approach is taken in MYCIN [Shortliffe76] but it requires such strong independence assumptions that it is only feasible in restricted domains. Another approach is taken by INTERNIST [Pople82], in which diagnoses are incrementally constructed by repeatedly choosing a disease that explains the most unexplained symptoms, until there are no unexplained symptoms left. While intuitively appealing, this does not guarantee coverage of the possible disease combinations. The approach used in

[Reggia83] addresses this coverage problem by considering every set of diseases whose combined symptoms cover all and only the observed symptoms. By Occam's razor, the hypotheses that should be considered are the minimal covering sets — those that do not include diseases not needed to explain the symptoms. Using probabilistic knowledge the likeliest of the minimal combinations is then chosen as the preferred diagnosis. Each of these approaches, however, perform poorly when the symptoms of the various diseases interact.

2.1.4 Summary of Symptom-Based Approaches

Work on symptom-based programs for automated diagnosis has yielded a number of powerful and useful techniques. These include (i) observation and test selection based on decision theory, with entropy as the heuristic evaluation function (ii) the use of causality as an organizing principle for diagnostic knowledge, and (iii) the formalization of diagnosis in terms of covering sets, allowing for diagnosis of multiple simultaneous diseases. The principal difficulty with symptom-based approaches is that the correctness and coverage of the knowledge base is difficult to guarantee, especially in the face of changes to the underlying system. When the available domain theory is weak, with only empirical associations between underlying diseases and observable symptoms, the symptom-based approach is reasonable and can be successful. Its limitations, however, motivate the model-based approach discussed below, which can provide better coverage and extensibility in domains where those properties are important.

2.2 The Model-Based Approach

Model-based troubleshooting is a widely investigated and well established methodology. The majority of the programs that share this paradigm are for diagnosis of designed artifacts such as circuits, so the term "device" will be used interchangeably with "system," and the notion of a "disease" will be replaced by that of a "fault." The key to the model-based approach is the representation of the structure and behavior of the correctly functioning device. This representation is used to make predictions about the behavior of the real device and about the outcomes of possible observations. Discrepancies between the predicted behavior and the actual observations are traced to sets of possibly malfunctioning components. Each set of components whose failure could explain the observations will be called a *candidate*; these candidates can be ranked according to their relative likelihood. As new discriminating observations are added some candidate will eventually dominate the others and be chosen as the final diagnosis, a set of components believed to be failing. With a hierarchic representation of structure, the isolation process can be repeated recursively on the substructure of each component believed to be faulty.

The key advantage of using knowledge about the correct behavior of components is that it dispenses with the need for storing associations between underlying faults and observed misbehaviors of the entire device. Instead, any subset of components whose predicted combined behavior disagrees with the behavior actually observed contains at least one broken component. By gathering more observations the troubleshooter can narrow down this set. Furthermore, this requires no commitment to the number of faults actually in the device, since the model can support reasoning about the interactions between any number of failing components. Finally, as noted earlier, when causal knowledge is available it can be easier to obtain than knowledge about overall associations between symptoms of failures and possible underlying faults.

It is useful to consider model-based troubleshooting in terms of four basic activities: *modeling*, *behavior prediction*, *candidate generation*, and *discrimination.* The following sections discuss each of these activities; a more complete survey appears in [Hamscher87].

2.2.1 Modeling

In model-based troubleshooting the notion of a "device model" is almost universally understood to mean a lumped element description, that is, the structure of the device is represented as a network of typed components and connections between them. Examples of models in various domains include:

- Circuit schematics with resistors, diodes, and so forth. This representation of analog circuits is used in INTER [deKleer76], WATSON [Brown76], SOPHIE [Brown82], IDS [Pan84], IN-ATE [Cantone83], DEDALE [Dague87], and others [Milne85].

- Circuit schematics using logic gates and higher-level digital components such as multiplexors and adders. This representation is used in HT [Davis84], DART [Genesereth84], and others [Friedman83] [AbuHanna88].

- Piping and instrumentation diagrams, which include components such as valves, potentiometers, lamps, and so forth, used in LES/LOX [Scarl85].

- Models of human physiology. Fluid models in terms of compartments, their permeable membranes, and so forth were used by ABEL [Patil81], by the system proposed in [Kuipers84], and by the Heart Failure Program [Long86]. A model of the human nervous system in terms of unidirectional neural pathways was used in LOCALIZE [First82].

The behavior of the entire device is taken to arise from the interaction of the behaviors of the individual components through the connections. Developing a particular description involves choosing a vocabulary of components and their behaviors, then representing the device as a connected network of these components. Therein lies a key advantage of model-based troubleshooting over traditional approaches: for designed artifacts, it can work directly from device models already developed for design and analysis. Model-based circuit troubleshooting, for example, can in principle work from ordinary circuit schematics and board layout information needed for design and manufacture. Therein also lie some of the deepest problems in the methodology: identifying the principles for building device models that are appropriate for model-based troubleshooting when the inherited models are inappropriate. Indeed, one of the reasons that there are relatively few projects using the model-based approach in medical domains is the scarcity of good analytic models for any substantial system.

The modeler confronts three goals simultaneously: achieving *fidelity*, *precision*, and *efficiency*. A model has fidelity when it does not support incorrect predictions about the device. A model has precision to the extent that the predictions it makes are strong enough to be falsifiable by observations of the actual device. A model is efficient when the work needed to make predictions using it is proportional to the benefits to be gained.

Fidelity is the primary modeling goal in troubleshooting. This is because if the model makes incorrect predictions, then discrepancies between the actual device and the model will be wrongly blamed on failures in the device. One way of ensuring fidelity is to (i) ensure that the primitive elements of the model support correct predictions about the corresponding primitive elements of the device when they are in isolation, and (ii) ensure that the ways in which the primitives can be composed preserves fidelity, just as the composition of the real elements of the device does not change those elements. This is the basic idea behind the principle of *no function in structure* [deKleer84]. No function in structure means that the description of a component behavior may not rely on the correct functioning of the whole device.

Ensuring that the models of primitive components are correct in isolation involves making sure that all the ways they could interact with other components are represented explicitly. For example, to say that "when the switch is closed current will flow" is incorrect because it neglects the fact that a voltage drop is required for current to flow. It also neglects to mention that if there is a temperature differential between its terminals there will be a conductive heat flow. Nor does it mention that if the switch were shorted to some wire elsewhere in the circuit then current could flow through that short. Any such interaction not represented in the model is a potential source of misdiagnoses, and the more interactions left out, the worse the problem.

Precision is another modeling goal. A trivial device model would make no predictions at all; it has fidelity, since it makes no false predictions, but it is useless for troubleshooting because it cannot produce any discrepancies either. A useful model produces predictions that can be confirmed or denied with the available observations.

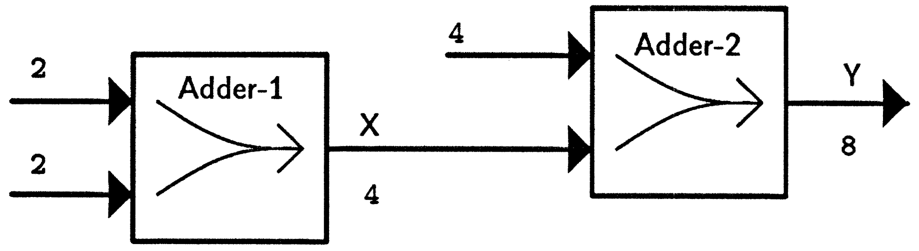
Finally, the more precise the model and the greater its fidelity, the less efficient it is to use. Consider simulating any substantial digital circuit with component models that included not only voltages and currents in the wires and transistors, but the temperature and specific heat of each contiguous piece of metal and semiconductor, the electromagnetic interactions with every other component, and so forth. A model with so much detail is obviously impractical and highlights the key dilemma for the modeler: how to sacrifice fidelity and precision in ways that gain efficiency. Of these, sacrificing fidelity is more serious, since it results in incorrect diagnoses, while sacrificing precision only results in ambiguity among different diagnoses. Interactions can be ignored for which only unlikely failures would make the interactions have noticeable effects. In the switch example earlier, being shorted to another wire in the circuit is possible and could have noticeable effects on the switch, but if shorts are unlikely failures in general it is reasonable to ignore that possible interaction in the switch model.

2.2.2 Behavior Prediction

The prediction task encompasses any categorical reasoning about the state of the device based on observations of its behavior. Given a device model built up as a network of components each with its own local behavior description, to a first approximation behavior prediction can be done by propagating many individual predictions local to each component.

For example, suppose both inputs to an adder component Adder-1 are believed to be 2 (Figure 2.1). The output of the adder can then be computed using only local knowledge about its intended behavior. Similarly, the output of Adder-2 can be predicted using its two inputs.

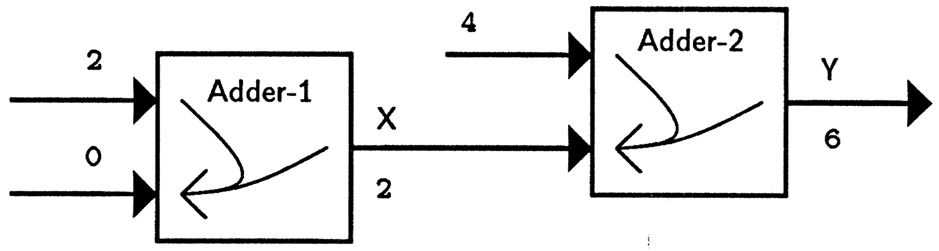
Figure 2.1: Behavior Prediction Example



Behavior prediction in that case is simply a kind of simulation: conclusions about the adder outputs were based on their inputs. However, the behavior model need not only predict outputs from inputs, but can enforce any logical relationship between the values carried by connections in the device. For example, if one input to Adder-2 is 4, and the output is 6, then the other input is predicted to be 2 (Figure 2.2). Similarly, if one input to Adder-1 is 2, then the other input is deduced to be 0.

The technique of predicting behavior by accumulating local predictions can be extended to reasoning about time-dependent behavior. For example, when all the inputs and initial state of a flip-flop are known over a certain interval of time then the outputs can be predicted over that interval

Figure 2.2: Reasoning from Effects to Causes



as well. The obstacle that behaviorally complex devices present is that in general this means explicitly computing and representing every event. In the flip-flop example, the events in question are changes of boolean value. Reasoning about the behavior of a digital circuit over any appreciable length of time is impractical; the culprit is the sheer number of clock transitions and consequent changes of state that might be involved. Devices with complex time-dependent behavior motivate the use of abstractions that allow predictions to be made without having to explicitly construct such extensive sequences.

For efficiency, the nominal behavior of the device given some standard stimuli may be stored as part of the model. In the model of human acid-base and electrolyte equilibrium in ABEL [Patil81], for example, each parameter of the model has an expected value assuming normal patient activity (for example, normal fluid intake). Similarly, the troubleshooting systems of [Cantone83] and [Milne85] store nominal values at circuit nodes for each of a fixed set of tests. This is at least a partial solution to the problem of expensive predictions. This thesis takes a different approach, focusing instead on having abstractions that will support economical prediction.

2.2.3 Candidate Generation

When discrepancies are found between the observed behavior and the behavior predicted by the device model, candidate generation produces one or more explanations for those discrepancies. There are at least three ways of approaching this task.

The first technique is to associate with each prediction made in the model the sets of components whose correct behavior would support that prediction. For example, the prediction in Figure 2.1 that the output of the second adder is 8 would be supported by the set of components {Adder-1, Adder-2}. With this supporting information, each discrepancy can be explained by the failure of one or more of the components in those sets. For example, if the output was observed to be 6 instead of 8, then at least one of those two components is broken. If there are several discrepancies, then the broken components must form a covering set (as in [Reggia83], where the symptoms of the diseases present must form a covering set of all observed symptoms). If a single failure is assumed, then the candidates form the intersection. There are differences in the machinery — especially in the way that dependencies between predictions and components that support them are recorded — but this idea is at the core of the candidate generation procedures in [deKleer76], [Brown82], [Davis84], [Genesereth84], [Scarl85], [deKleer87], and [Dague87]. The details of GDE [deKleer87] will be presented shortly, since it provides the basis for the program in this report. [Ginsberg86] and [Reiter87] provide formal interpretations for this technique based on the notion that broken components are abnormal and the preferred diagnoses are those requiring the minimal abnormalities. An important advantage of this technique is that it requires no information about how certain components might fail; only the correct behavior needs to be known.

A second technique extends the first by taking advantage of fault models — knowledge about how individual components fail. After finding components whose failure could explain all discrepancies, the effects of known failure types in those components are simulated. If the set of known failures is treated as exhaustive, then candidates can be exonerated by fault simulation. For example, suppose some wire is a candidate. Wires fail only by breaking, so the program could simulate the effects of that wire becoming an open circuit and check whether that is consistent with the observations. If it is not consistent, the wire would be exonerated. This technique is used in SOPHIE [Brown82] and several other model-based troubleshooting programs. IDS [Pan84] goes further and explicitly models component failures in a way that allows dependent failures — failures caused by prior failures — to be explicitly represented and diagnosed. The additional power that fault models provide, however, comes at a high price, since it is difficult to provide an exhaustive list of failures for anything other than the simplest of components.

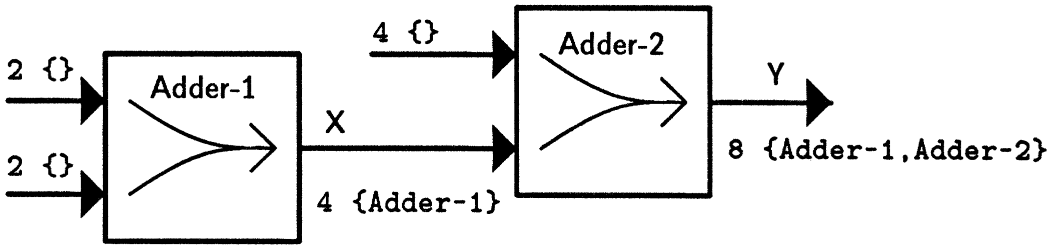
The third technique generates alternative explanations for each discrepancy incrementally, as in ABEL [Patil81]. For example, if in Figure 2.1 the output had been observed to be 6 instead of 8 as expected, among the initial possibilities are that Adder-2 is broken, that one or both of its inputs are lower than expected, and that one of the inputs is higher than expected and the other lower. Some of these are inconsistent with the observations (for example, one of its inputs is known to be 4) and are discarded; the others survive to be further elaborated. The knowledge about the system is the same as that available to the previous technique; the difference is that generating candidates and using fault models to check their consistency is interleaved. The advantage of doing so becomes evident when diagnosing a system with feedback or with high connectivity between its components. If only knowledge about correct behavior is used then almost any discrepancy can be accounted for by the failure of any component [Hamscher84]. Subsequent reasoning with fault models can constrain the possibilities, but it is inefficient to go through the intermediate stage of generating all possible candidates, and the interleaving avoids it.

The program described in this report is based on the first of the above techniques, as implemented in GDE [deKleer87]. This approach begins with an augmentation of behavior prediction. Each local prediction is tagged with the set of components on whose correct behavior it depends, so that when an observation is made that contradicts what the model predicted, the components responsible can be easily found. Each of these predictions are only valid if one or both adders are assumed to be working normally, and each prediction is tagged with the minimal sets of assumptions that support it. For example, suppose both inputs to an adder component Adder-1 are 2 (Figure 2.3).

Neither input to Adder-1 requires any assumptions, so their tags are {}. The prediction that the output X is 4 relies on the assumption that Adder-1 is working normally along with all assumptions supporting the inputs, so it is tagged with the set {Adder-1}. Each such set of assumptions is called an *environment.* The prediction that the output Y is 8 is tagged with the environment containing the assumptions that Adder-1 and Adder-2 are both working. Observations such as those at the inputs of Adder-1 are true in the empty environment since they rely on no assumptions.

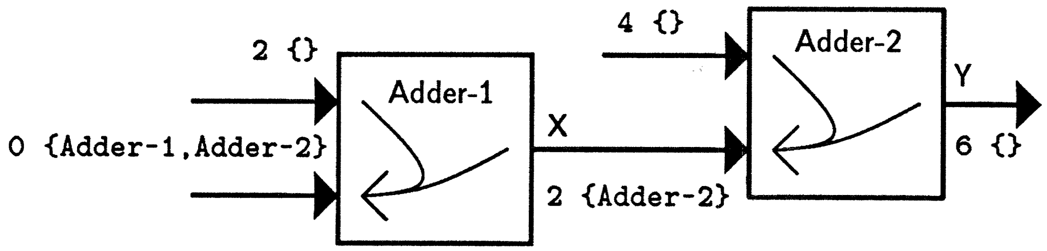
Recall that the behavior model need not only predict outputs from inputs, but can enforce any logical relationship between the values carried by con-

Figure 2.3: Behavior Prediction Example



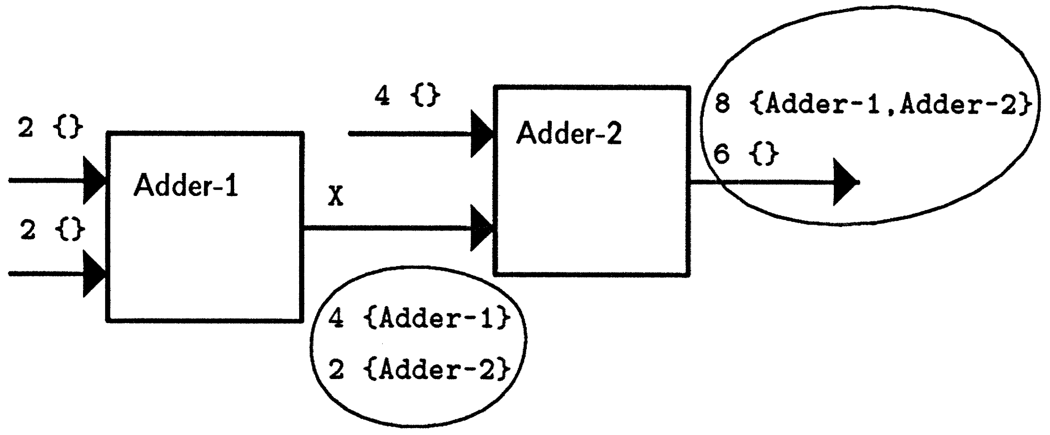
nections in the device. Such predictions are tagged with sets of assumptions just as before. For example, if one input to Adder-2 is 4, and the output is 6, then the other input is predicted to be 2 and tagged with the assumption that Adder-2 is working (Figure 2.4). Similarly, if one input to Adder-1 is 2, then the other input is deduced to be 0 and that prediction is tagged with the assumptions that Adder-1 and Adder-2 are working.

Figure 2.4: Reasoning from Effects to Causes



Candidate generation involves detecting discrepancies and determining which components could have been responsible. Discrepancies are inconsistent predictions made under different sets of assumptions (that is, in different environments). For example, suppose the inputs to the two-adder device were as in the first case, but the output was observed to be 6 (Figure 2.5). Superimposing the two sets of predictions, it can be seen that (among other discrepancies) node X is predicted to be 4 if Adder-1 is working, but 2 if Adder-2 is working. The union of the environments that underly inconsistent predictions are termed *conflicts*, and are denoted with angle brackets ⟨ ⟩. In this case, ⟨Adder-1, Adder-2⟩ is a conflict.

Figure 2.5: Discrepancies Produce Conflicts



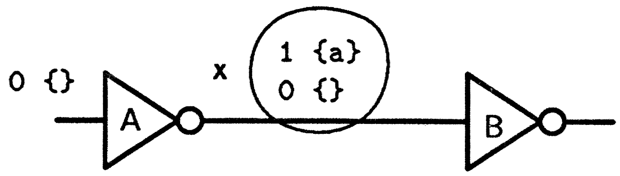
A conflict is a set of assumptions that contains at least one that must be false. In troubleshooting, the assumptions are about whether components are working properly, so it can be thought of as a set of components that cannot all be working properly. If one of the components in each conflict were actually failing, it would resolve the inconsistency. The minimal set covers of these conflicts are termed *candidates*, denoted with square brackets [ ]. By Occam's razor only the minimal set covers (those with no subsets that are covers) are needed; the minimal covers are the simplest explanations for the inconsistency. Each candidate corresponds to a set of components that would resolve all the inconsistencies if all of them were failing. For example, if there is just one conflict ⟨Adder-1, Adder-2⟩ there are two singleton candidates, denoted [Adder-1] and [Adder-2]. The covering set that includes both adders is not a candidate, since it is not minimal.

This scheme incorporates the handling of multiple faults in a natural way. Suppose we subsequently observe that X is 5. There would then be two conflicts ⟨Adder-1⟩ and ⟨Adder-2⟩, and their minimal set cover would be the candidate [Adder-1, Adder-2], meaning that both Adder-1 and Adder-2 are faulty. In general, the number of candidates can be exponential in the number of conflicts. Consider for example $2n$ assumptions and n conflicts, one for each pair of assumptions $2i$ and $2i + 1$; this results in 2^n candidates. Exponential blowup is rare in practice; a more common phenomenon is that along with a small set of single-fault candidates there will be a larger set of multiple-fault candidates. For example, the two conflicts ⟨A, B, C, D⟩ and ⟨D, E, F, G⟩ yield one single-fault candidate [D] and nine two-fault candidates.

Strictly speaking, this and other model-based schemes do not do diagnosis. They detect differences between the device and the model and produce candidates that indicate which components of the model could be modified to account for the observations. To interpret these differences as indications that certain components of the real device are broken requires that the model has fidelity, that is, that the models of components accurately represent their correct behavior. Because of the practical impossibility of having models that are correct in every respect, it is important to understand how GDE degrades in the face of failures whose effects are not properly modeled. The central issue is which interactions between components have been modeled; failures that result in the coupling of components through unmodeled interactions will yield incorrect candidates.

For example, the standard model of digital circuits says that each node is driven to 0 or 1 by just one gate. Using this model, upon finding a discrepancy at a given node, only the gate driving the node and gates upstream from it will appear in the resulting conflict (Figure 2.6).

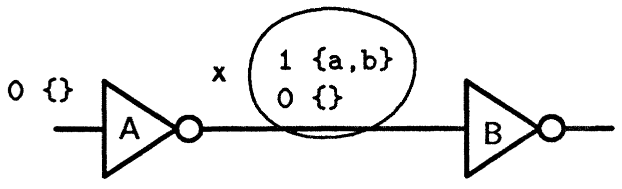
Figure 2.6: If x is not 1, Only A Could be Broken



In reality, the gates also interact through current flow. The gate B being driven could be failing in a way that pulls down the node x. The inverters are coupled via an interaction that was not modeled, so the standard digital model yields the wrong answer. Suppose current flow were modeled, so that the node x is 1 only when both A and B are working (Figure 2.7).

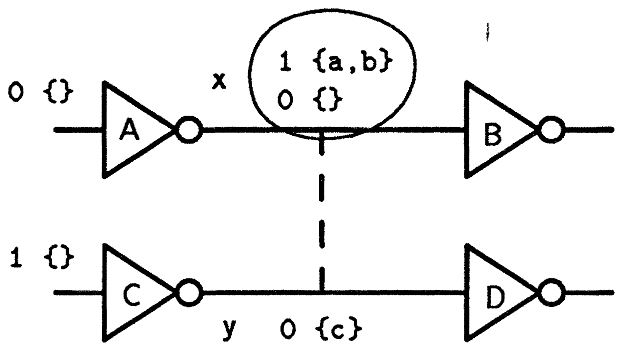
Now both inverters will show up as candidates. These candidates could be disambiguated by observing the current flow into B.

Figure 2.7: Inverter B Could be Pulling x Down



There are even more candidates, however. For example, there could be a short between node x and some other node (Figure 2.8). Even if modeling all the possible shorts were practical, there are still further possible interactions that this model leaves out.

Figure 2.8: A Short Could be Pulling x Down



An important property of the GDE scheme for producing fault candidates is the way it degrades in the face of failures that violate the device model by introducing unexpected interactions, as in this last example. With enough observations, the results will be interpreted as requiring multiple-fault explanations. For example, suppose the symmetric test had been run with the inputs to A and C swapped, and y observed to be 0 instead of 1. A new conflict ⟨C, D⟩ would have been discovered, and GDE would produce four candidates: [A, C], [A, D], [B, C], and [B, D]. Among the candidates that GDE produces will be multiple-fault candidates involving the components influenced by the new connection. In fact, any failure can be interpreted as a multiple-fault failure no matter how drastic its effects, since there is always the degenerate candidate consisting of all the components in the device. For example, suppose that some digital circuit model does not explicitly represent power and ground. If power were lost then every component might appear to be behaving incorrectly. That is exactly what the troubleshooting engine would produce as a candidate: one in which every component is broken.

The GDE scheme for generating candidates from conflicts is simple, general, and to the extent that the model accurately represents the structure and behavior of the device it yields correct results. The difficulty is that since the model can never be totally correct, only the fidelity of the underlying model gives license to interpret a candidate such as [A, B] as meaning that both A and B are broken. One way of dealing with this problem is illustrated by HT [Davis84], in which discrepancies that can only be explained by multiple fault candidates are checked to see whether they could be explained as single faults in alternative models of the circuit. One such alternative model makes the physical proximity of wires explicit, to detect shorts like that in Figure 2.8.

Another difficulty is one shared by any troubleshooting program, namely, the available observations of the device might be too crude to detect discrepancies. For example, suppose a behavior model predicts a particular sequence of zeroes and ones will appear on a wire, but an oscilloscope can only determine whether the signal is active or not. Legitimate discrepancies and conflicts may well go undiscovered, and hence some inconsistent candidates may survive.

2.2.4 Discrimination

As diagnosis proceeds there are usually several candidates that could explain all the discrepancies. To discriminate between these candidates requires gathering more information in the form of either (i) new observations of the device in its current state, or (ii) observations of its response to some new test stimuli. Since there are typically many observations and tests that could be performed, the program needs to choose which of them to do next. This choice can be formulated in terms of the cost of each action, the benefits of their various outcomes, and the likelihoods of those outcomes. Using the entropy of the distribution of candidates as a "benefit" metric, choosing the observation yielding the minimum expected entropy as in [Gorry73] can be used in the model-based approach just as in the symptom-based approach. In GDE the device model is used to derive the expected outcomes of each possible observation along with their likelihoods; the details will be discussed shortly. A similar framework is used in IN-ATE [Cantone83] to estimate the likelihoods of various circuit test outcomes.

Recall that in GDE each candidate is a set of assumptions that would resolve all conflicts if they were all false. GDE assigns a weight to each candidate by treating each assumption as independent and assigning to each a prior probability near 1.0 of being true. The probability of a candidate is then the probability that all the assumptions it includes are false and all other assumptions are true. The weight of each candidate is its probability normalized with respect to all candidates. Continuing the two-adder example, let the initial probability of each adder working be $p(\text{Adder}) = .99$. The weight of each is .50, computed as shown:

Candidate	Probability	Weight
[Adder-1]	$(1 - p(\text{Adder-1})) \times p(\text{Adder-2}) = .0099$	.50
[Adder-2]	$p(\text{Adder-1}) \times (1 - p(\text{Adder-2})) = .0099$	.50

Suppose there had been three adders A, B, and C with $p(A) = p(B) = p(C) = .99$, and that there were two conflicts $\langle A, B\rangle$ and $\langle B, C\rangle$. There would be two candidates [B] and [A, C] whose rankings would be as shown below[1]. This yields the intuitively satisfying result that the single-fault candidate [B] is much more likely than the multiple-fault candidate [A, C]:

Candidate	Probability	Weight
[B]	$p(A) \times (1 - p(B)) \times p(C) = .0098$	.99
[A,C]	$(1 - p(A)) \times p(B) \times (1 - p(C)) = .000099$	.01

There will nearly always be several competing candidates. To discriminate among them, GDE considers all the possible observations that could be

[1]The normalization is a heuristic step that ignores non-minimal candidates. Both A and B could be broken, all three could be broken, and so forth. The residual probability is distributed among these other non-minimal candidates.

made next, and by a one-level lookahead picks the observation that is expected to yield the most information. The probability of each outcome of a possible observation is estimated as the combined weight of those candidates with which the outcome would be consistent.

In the two-adder example, according to the predictions an observation at X has two outcomes; either it is 4 (if Adder-1 is working), or it is 2 (if Adder-2 is working). An outcome of 2 is consistent with the candidate [Adder-1], and 4 is consistent with the candidate [Adder-2]. Each candidate weight is .5 so the probability of each outcome is estimated as .5 as well. The expected information gain from making a given observation can be estimated as the negative of the entropy in that distribution of outcomes (the sum of $p_i \log_2(p_i)$ over the outcomes i). The observation that maximizes the additional information is selected. In the two-adder example the computation is trivial. The entropy is $.5 \log_2(.5) + .5 \log_2(.5) = -1.0$ and the information is $-(-1.0) = 1.0$. A probe anywhere already observed yields information of 0 and X is the only signal not observed, so probing X is obviously the right choice. In less trivial examples this technique tends to choose observations that, roughly speaking, divide the space of outstanding candidate weights in half.

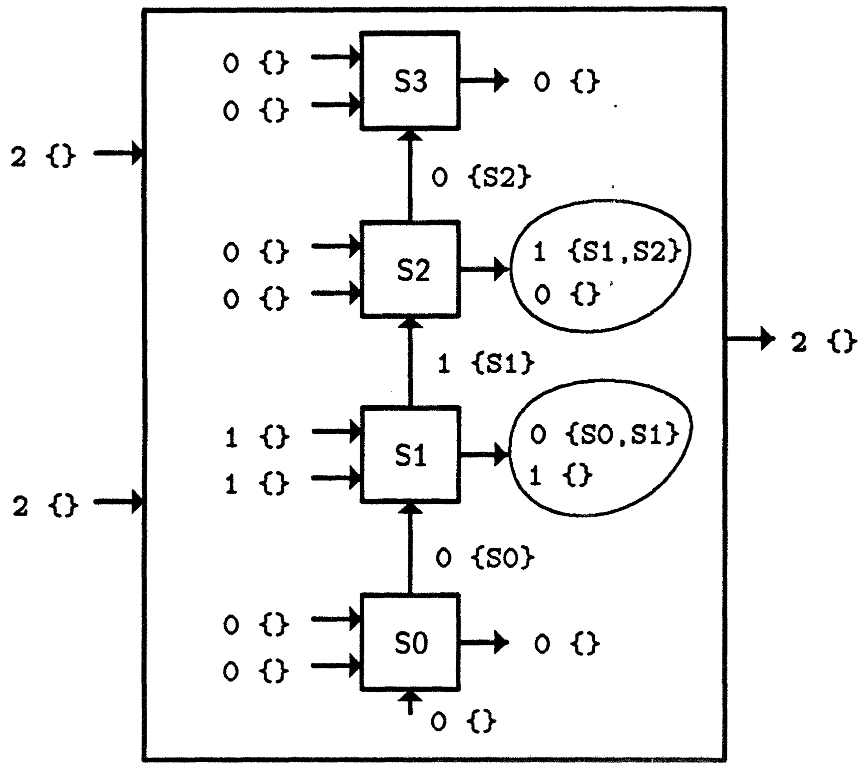
Relying on a fixed set of observations or tests is not always practical, however. In domains such as digital circuit diagnosis it can be more effective to design a test specifically to help discriminate between candidates. This approach is taken by DART [Genesereth84], which repeatedly generates tests (using an implementation of path sensitization [Roth67]) until it finds one that will yield distinguishable outcomes given different candidates. Such tests can be generated more effectively if information about candidates is used while creating the test [Shirley83]. The program discussed in this report selects observations based on the scheme in GDE, but neither selects nor generates tests.

2.2.5 Hierarchic Diagnosis

Hierarchic diagnosis is usually viewed in terms of recursive descent. The troubleshooting program first isolates the fault to a component at a certain level of detail, then proceeds to diagnose the failure within its substructure, until a primitive level of detail is reached. Each level of structural detail usually has associated with it a level of behavioral detail as well. Nearly all model-based troubleshooting programs incorporate hierarchic diagnosis controlled in this way.

The GDE scheme can be extended to do hierarchic diagnosis. For example, suppose in the two-adder example that X is observed to be 2, so that ⟨Adder-1⟩ is the only conflict and hence [Adder-1] the only candidate. If the adders are not primitive components, but rather have the substructure of four-bit ripple-carry adders (Figure 2.9), then troubleshooting can continue at the structural level of full adder slices and behavioral level of bits. Each of the adder slices has a "sum" bit output and a "carry" bit output that feeds into the next slice.

Figure 2.9: Diagnosis of Adder-1



The model predicts that if S0 is working its carry output will be 0 . The sum output of S1 is predicted to be 0 if both S0 and S1 are working. The carry output of S1 will be 1 no matter what the carry-in from S0 was, since two of its inputs are 1 already. The sum output of S2 will be 0 if both S1 and S2 are working. The observation that the adder output is 2 corresponds to observations that the sum outputs of S0 through S3 are 0, 1, 0, and 0 respectively. These observations are inconsistent with the outputs at S1 and S2, producing two conflicts ⟨S0, S1⟩ and ⟨S1, S2⟩. These two conflicts yield the single-fault candidate [S1] and the two-fault candidate [S0,S2].

Note that hierarchic diagnosis can also be worthwhile even when the fault has not been fully isolated (technically, isolation would mean that the assumption that the component is working is a singleton conflict). In the two-adder example, suppose that X has not yet been observed, so that both [Adder-1] and [Adder-2] are candidates. Both adders are descended into, revealing slices S0 through S3 in Adder-1 and S4 through S7 in Adder-2. Some of the newly discovered conflicts are shown in Figure 2.10: ⟨S1, S2, S3, S6, S7⟩, ⟨S0, S1, S2, S5⟩, ⟨S1, S2, S4, S5⟩, and ⟨S0, S1, S4, S5⟩.

From these conflicts all of the subcomponents of Adder-2 can be ruled out as single-fault candidates, without requiring any more observations. In fact there is only one singleton candidate: [S1]. The other minimal candidates are [S0, S2], [S2, S5], [S2, S4], [S3, S5], [S5, S6], [S5, S7], [S0, S4, S7], [S0, S4, S6], and [S0, S3, S4].

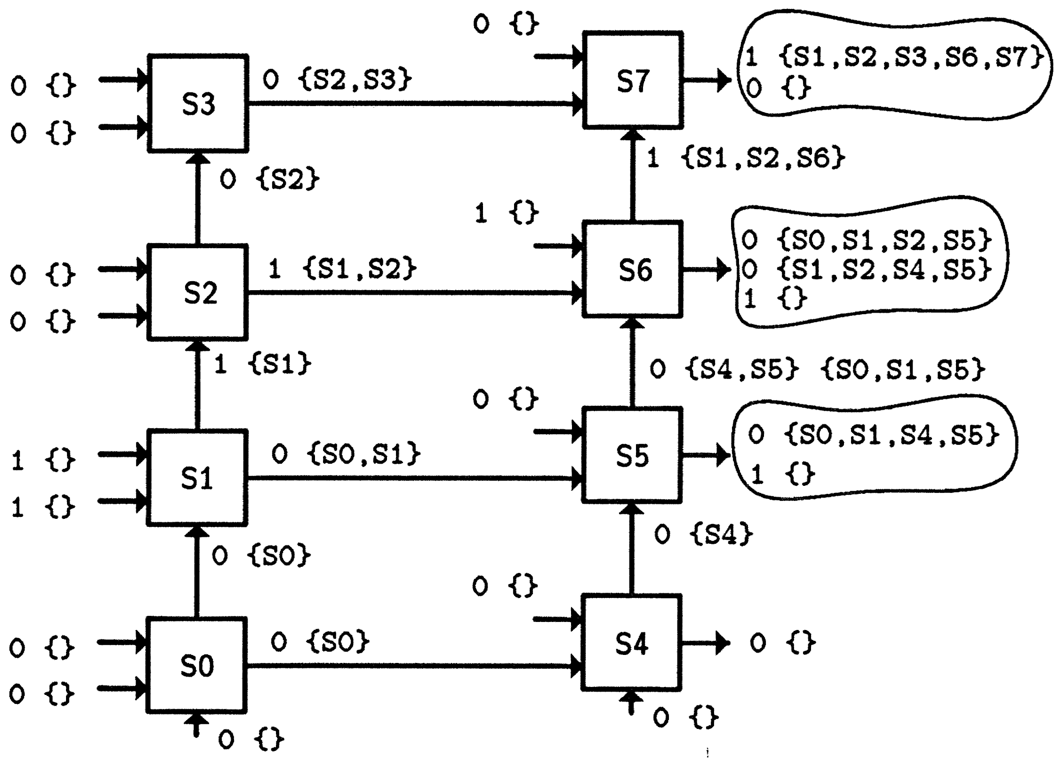
Most discussions of hierarchic diagnosis in model-based troubleshooting programs present a simplified picture in terms of isolation to a single component followed by recursive descent. As this example suggests, effective diagnosis of more complex systems is likely to require considering multiple levels of detail even when there are several candidates, as done in ABEL [Patil81] and in the program discussed in this report.

2.2.6 Summary of the Model-Based Approach

Although differing in implementation technology, all model-based troubleshooting programs share the same underlying organization. A device model produces predictions about behavior and about what ought to be observed. A separate troubleshooting engine then produces alternative diagnoses that each resolve all discrepancies between the model and the actual observations.

The notion of a "device model" is that of a lumped-element description consisting of components and connections. In committing to any such repre-

Figure 2.10: Diagnosis of Adder Substructures



sentation the program sacrifices some degree of coverage, since there will be failures that it will misdiagnose.

Behavior prediction in such a model can (for the most part) be done by local propagation, that is, each prediction is made on the basis of information local to a single component. The choice of level of detail to represent behavior and of the machinery that manipulates it both inevitably sacrifice precision and completeness of predictions for the sake of efficiency. In troubleshooting, the effect is to sacrifice some degree of resolution since there will be some failures that cannot be distinguished.

In the GDE framework, each prediction is tagged with its set of supporting *environments* — sets of assumptions about which components are working. Discrepancies result in *conflicts* — sets of assumptions that contain at least one false assumption. Each covering set of these conflicts is a possible diagnosis; by Occam's razor the minimal covers are selected as *candidates.* One of the important properties of the scheme is that when faced with a failure that cannot be represented in the model, it proposes multiple-fault candidates rather than (say) declaring an irreconcilable inconsistency.

There are nearly always several different candidates. Candidates are discriminated by assigning each a weight based on its normalized prior probability, and if there is no clearly dominant candidate, an observation with the minimum entropy is selected. When further conflicts result from the observation, some candidates are ruled out and others become more likely.

Finally, having isolated a fault to a single component, hierarchic diagnosis proceeds by descending into substructure of the component, if any. The additional information available at lower levels of detail may also be useful for discriminating candidates even if the fault has not been uniquely isolated, as illustrated above.

Chapter 3

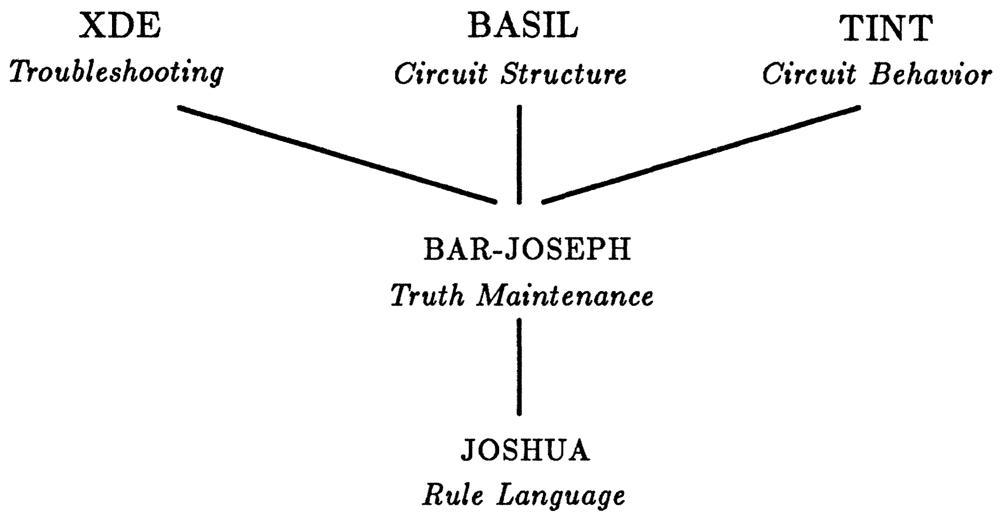
Troubleshooting Scenarios

This work makes several claims about representing digital circuits for model based troubleshooting. The support for these claims comes largely from a set of implemented examples of circuit structure and behavior, and from the fact that the troubleshooting engine can successfully diagnose faults using those models. The scenarios have been collected into this early chapter to provide context and motivation for subsequent discussions of the structure and behavior of these circuits. Indeed, a central theme of this work is that the intended use of a model impacts what gets mentioned in the model; this chapter shows the reader that intended use.

The program that does these examples is organized into several subsystems (Figure 3.1). There is a domain independent troubleshooting engine XDE that extends the GDE approach so as to use hierarchic diagnosis and fault models. The physical and functional organization of the circuits to be diagnosed are represented in a language called BASIL. The behavior of the components in those circuits are represented in a temporal constraint propagation language TINT. All of these are built using JOSHUA [Rowley87], which provides implementations of data storage and retrieval along with forward and backward chaining rules. BAR-JOSEPH embodies the author's extensions to JOSHUA, including a simple inheritance facility and an assumption-based truth maintenance system based on boolean constraint propagation [McAllester80b] [deKleer86a]. Chapters 4 through 7 discuss XDE, BASIL, and TINT; the underlying JOSHUA and BAR-JOSEPH implementations are not discussed in detail.

The troubleshooting examples are all taken from the Console Controller

Figure 3.1: Overall Troubleshooting Program Organization



Board of the Symbolics 3600 series console. The board has approximately 50 chips and 300 visible circuit nodes; the largest example currently handled involves 20 chips and 100 visible nodes. In the descriptions of structure and behavior, the following conventions are adhered to:

- U25 is a typical chip name. RN7 is a typical name for a nine-resistor network that is treated just like a chip.
- n178 is a typical name for a circuit node, or, to be precise, for a wire etch as represented by the programs.
- FD01 is a typical name for a component such as a Frequency Divider. One-of-a-kind components are usually given one or two letter names such as U (a microprocessor) or R (the Reset Hold Counter).
- U30a and U30b are typical names for the flipflops that reside on chip U30. In general the a, b, c suffixes denote functional units within a chip.

The figures that show the physical and functional organization of circuits obey the following conventions:

- A box with thin lines indicates the boundaries of a physical component, usually a chip.
- A box with thick lines indicates a functional component such as a flipflop, which may have a complex correspondence to a physical component.
- Where a box name such as U is not sufficiently informative, the type of the box is shown in a slanted font as *Input Processor*.
- Thick lines with arrowheads indicate connections between components; technically they are "signals" as defined in Chapter 5.

The examples summarize the output transcripts found in Appendix A.1 through Appendix A.11:

- One example involving three chips in the section of the board responsible for generating clocks at various frequencies.
- Four examples involving ten chips in the Audio Decoder section, responsible for translating an asynchronous digital audio signal from the host into a signal that drives a speaker.
- Two examples involving twenty chips in the Input Encoder section, responsible for transmitting keystrokes and mouse motions to the host.

3.1 Clock Generator Examples

The Clock Generator circuit shown in Figure 3.2 and Figure 3.3 is responsible for generating 10 Mhz, 5 Mhz, and 2.5 Mhz clock signals that will be distributed throughout the board. It is a trivial circuit, of course, but nevertheless raises important issues.

The generator consists of a crystal oscillator OSC that produces a 10 Mhz TTL clock. The inverter in this circuit is acting as a buffer (FB01); the frequency of its output is the same as its input. Two separate frequency dividers FD01 and FD02 are implemented with the dual flipflops on chip U30; if all the components {U25,U32,U30} are working then the output at n158 is a 5 Mhz clock and the output at n167 is 2.5 Mhz.

Figure 3.2: Clock Generator Schematic

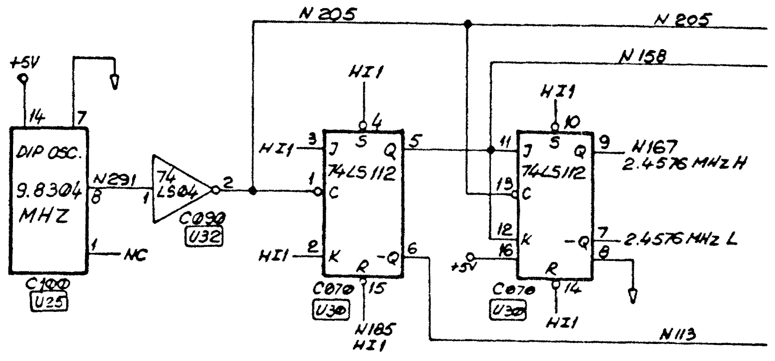
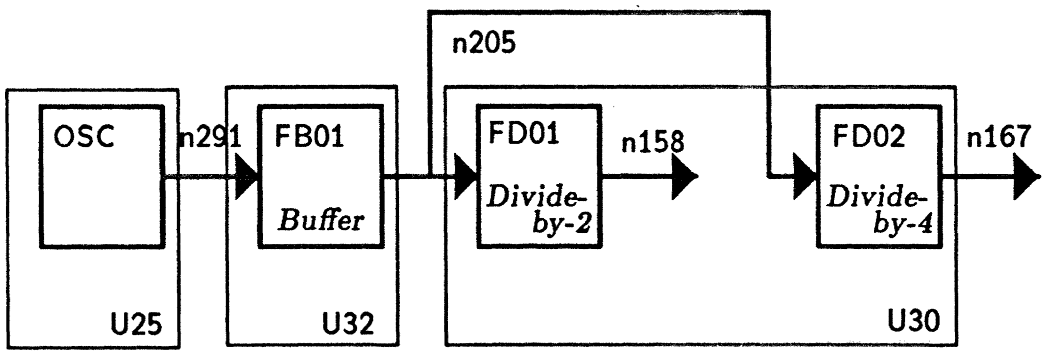


Figure 3.3: Clock Generator Structure



3.1.1 Troubleshooting the Clock Generator

Assume n167 is observed to be "flat," that is, its frequency is zero. This yields ⟨U25,U32,U30⟩ as a conflict and hence [U25], [U25], and [U25] as candidates. Crystal oscillators, because of their internal structure and the way they are packaged, tend to fail more frequently than other components. Hence U25 is more likely to be broken than U32 or U30. A probe at n291 can be made to confirm this.

Assume the signal at n291 is observed to have frequency 10 Mhz, as would be expected if the oscillator were behaving normally. Oscillators also fail in a characteristic fashion: they produce a flat output rather than the desired periodic waveform. Hence the oscillator becomes a much less likely suspect, though still logically possible since the exact shape of every pulse cannot be examined.

This leaves U30 and U32 as the main suspects and node n205 as the next good place to probe, because that would tell which chip needs replacing. If that signal is probed and observed to be flat (zero frequency) it is relatively certain that U32 is broken, since otherwise the signal would have had frequency 10 Mhz.

3.1.2 Morals of the Clock Generator Example

Simple as it is, the Clock Generator example illustrates three key ideas:

Temporally coarse behavior models can be adequate for troubleshooting. Although the Clock Generator is a digital circuit, the traditional model of digital behavior that involves individual clock cycles, rising and falling edges, and so forth, was inappropriate for this troubleshooting example. A much simpler, temporally coarse description of its behavior involving the notion of "frequency" provided just as much ability to localize the fault. The detailed model would have uselessly predicted many events individually undetectable using an observation technology as simple as an oscilloscope. That abstractions simplify reasoning is obvious, what is important is that in this case the nature of the abstraction was explicitly temporal and made traditional simulation unnecessary.

The representation of physical organization is essential for troubleshooting. Failures and repairs occur in physical devices, not in the functional organization that we attribute to them, hence the physical organization of the device needs to be represented explicitly. The value of representing the physical organization has previously been associated with the diagnosis of unusual faults such as solder bridges [Davis84]; in fact the model should always include the physical information, for the more mundane reason that it can save the troubleshooter from spending effort on distinguishing faults that share the same repair. In this example, there was no need to distinguish which of two flipflops might have been broken, since the repair in both cases was identical: replace chip U30.

Fault models are useful heuristics. There was added focusing power available from heuristic knowledge about relative failure rates of components and likely misbehaviors. In this example, without knowing that oscillators commonly fail in a particular way, the observation that the oscillator output had frequency 10 Mhz would have told us nothing at all about the oscillator. This kind of knowledge can only be used to discount possible diagnoses, never to support them directly. The added knowledge discounts the possibility that the oscillator was broken and hence promotes the more likely diagnoses. Conversely, had n291 been flat instead of 10 Mhz, the conflict ⟨U25⟩ would have resulted and the oscillator been identified as faulty after only one probe.

3.2 Audio Decoder Examples

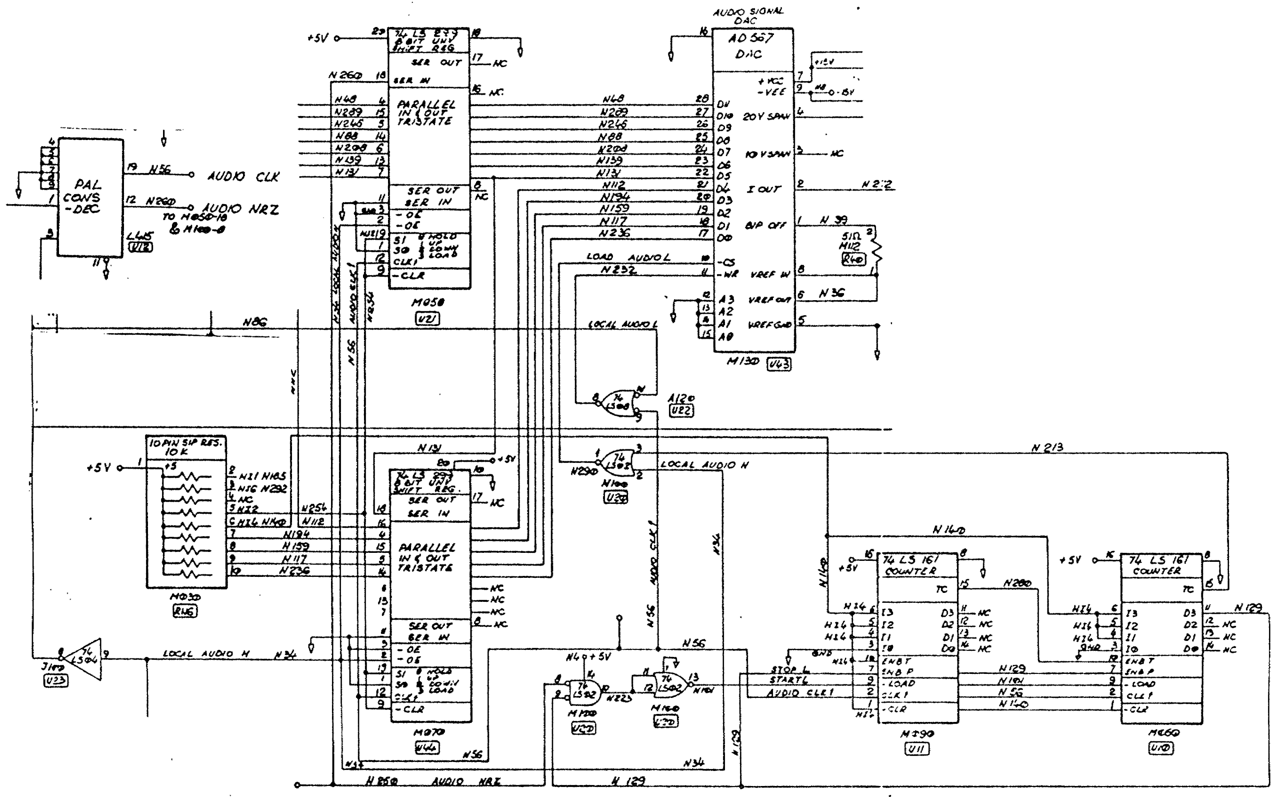
The Audio Decoder is responsible for converting an asynchronous serially encoded 12-bit digital signal into a voltage in the range +15 to -15 volts. It involves ten chips and fifty visible circuit nodes (Figure 3.4). The simplifications made for presentation are that explicit information about wire etches, and an alternate signal path into the analog-to-digital converter, have been omitted.

The four troubleshooting examples illustrate the following ideas:

The behavior of components should be represented in terms of features that are easy for the troubleshooter to observe. The vocabulary of observations that the troubleshooter can make provides a vocabulary that can be used in modeling the behavior of the device. For example, if one assumes that only certain features of a signal can be observed using an oscilloscope, then that set of features defines one level of abstraction at which to model the behavior of the device and its components. This model may not provide sufficient resolution, so a more detailed model may be needed as well; the point is that the vocabulary of observations provides guidance as to what abstractions may be useful.

Components in the representation of the functional organization of the circuit should facilitate behavioral abstraction.. Representing the organization of a device hierarchically has advantages noted earlier. Hierarchic organization by itself, however, provides no leverage on the fundamental goal of troubleshooting — to discriminate between candidates — unless there is a behavioral characterization of each component that would be difficult or

Figure 3.4: Audio Decoder Schematic



expensive to derive from its subcomponents.

3.2.1 Functional Organization of the Audio Decoder

Figure 3.5 shows the three stages of the Audio Decoder: a clock is extracted from the incoming asynchronous manchester signal by MTS01; the resulting clocked serial signal is converted into a 12-bit parallel signal with a write strobe by STP01; and the parallel signal is then converted to a voltage by the digital-to-analog converter PTA01. STP01, which converts from synchronous serial to parallel data, has three components: CSA01 accumulates the data bits in two shift registers, while a pair of counters in CSB01 count the number of bits since the first arrived. When all the bits have arrived CSB01 asserts n290 to latch the parallel data into the digital-to-analog converter. BUF01 buffers the serial clock n34 extracted from the incoming signal and strobes MTS01 using n232.

Most of these functional components can be viewed as simply converting information from one encoding to another. In particular, the signals denoted ser01 and par01 both carry streams of 12-bit digital values; only their underlying encoding is different. Hence, MTS01, STP01 and PTA01 are modeled abstractly as buffers. The burst detector CSB01 converts incoming 12-bit bytes into single pulses on its output. The "clocked serial accumulators" (CSA01, CSA02, CSA03) are shift registers that accumulate the incoming serial data bits in each burst. The individual data values are not represented explicitly. Rather there are abstract signals which, although in principle could be computed at every point in time, in fact are only observed and reasoned about in terms of features such as their amplitudes and rates of change.

Each of the signals shown is described using features that an oscilloscope can easily detect. An oscilloscope can be used to measure the frequencies and periods of signals. For nonperiodic signals, this can only be done qualitatively: a signal which is neither constantly high nor constantly low has some unspecified positive frequency, and is characterized as "changing." For periodic signals, certain shape properties can be observed: in particular, the difference between the maximum and minimum value, the period of crossings of its midpoint value, and the frequency of crossings of zero in the first derivative (that is, changes of direction). In these troubleshooting examples the Audio Decoder is presented with a 1 Khz sinusoidal signal[1]. The sampling rate is forty per period, that is, a new 12-bit quantity arrives every 25 μsec. The resulting digitally-encoded sinusoid is shown at the top of Figure 3.6. If this sinusoidal signal has higher order harmonic components, it is simply characterized as having a higher frequency in the first derivative than in the signal itself. The bottom of Figure 3.6 shows an example, a distorted sinusoidal voltage signal in which the frequency of sign changes in the first derivative is higher than the frequency of sign changes in the voltage.

[1]It is assumed that the 1Khz signal is the only test input available. It turns out, in fact, that other test inputs would not provide appreciably more diagnostic resolution given the observability constraints already assumed, so that in itself is not a major handicap.

Figure 3.5: Audio Decoder Functional Organization

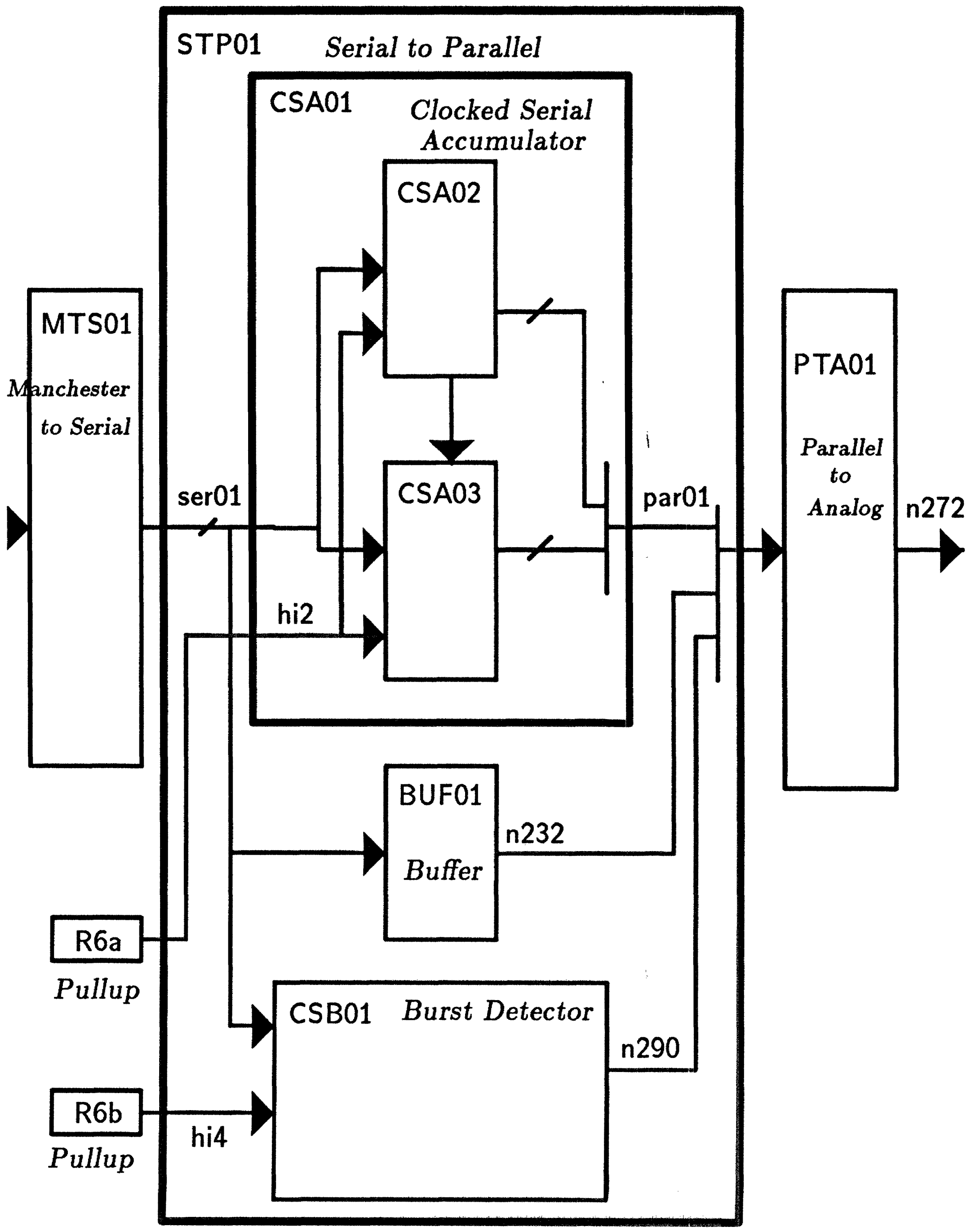
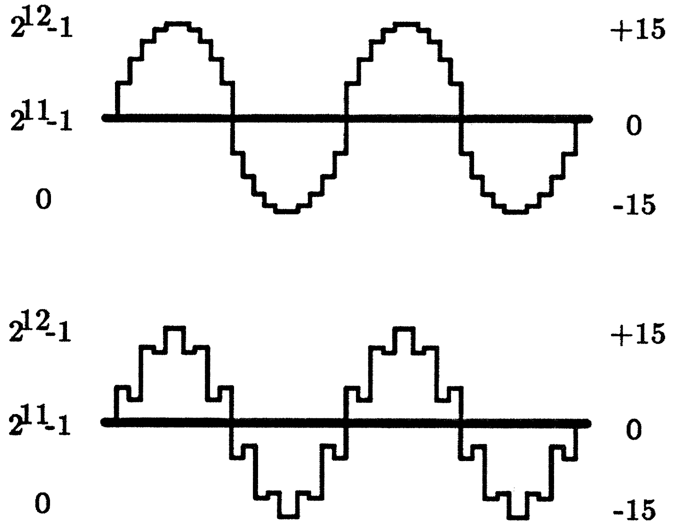


Figure 3.6: Signal with Too Many Zero Crossings in its First Derivative



The limited ability of the oscilloscope to characterize the voltage output of the Audio Decoder means that the signal **par01** need only be characterized in the vocabulary of the oscilloscope. Only the frequency of the signal, crossings of its midpoint value, and zero crossings in the first derivative, need be mentioned. Since the information carried by **par01** is encoded as a twelve-bit digital signal and cannot be directly observed, it is necessary to characterize the relationship between **par01** and the underlying signals that *can* be observed, namely the individual data bits and write strobes. To take two representative examples of this relationship: if the signal **par01** crosses its midpoint value with a frequency of n, then the most significant data bit has frequency of at least n because it has to change its value at least as often as **par01** does; similarly, if the write strobe signal is always high, then the signal **par01** never changes, so that its frequency is zero and the difference between its maximum and minimum is zero.

The accumulators **CSA01**, **CSA02**, and **CSA03** all act as delay elements: each incoming data bit appears some time later at each of the output bits of the shift registers. Hence given sufficiently many bytes transmitted, the frequency of each individual bit of the output signal should be the same as that of the incoming serial data measured with respect to the serial clock. To see why this is so, consider an 8-bit shift register that has an incoming signal clocked into its most significant bit. Suppose that input signal goes from 0 to 1 and back 1000 times during a certain time interval. The most significant output bit will change either 999 or 1000 times, the next-to-most significant output bit will change between 998 and 1000, and so forth. For a sufficiently large number of these cycles, the number of changes over that time interval are essentially equivalent, hence their frequencies are equivalent. This is an example of representing the behavior of components in terms of features that are easy for the troubleshooter to observe, in this case, in terms of whether or not the signals are changing.

The subcomponents CSA02 and CSA03 of CSA01 are almost identical to CSA01, except that CSA02 corresponds to chip U21, which holds the 7 most significant bits, and CSA03 to U44, which holds the 5 least significant.

The burst detector CSB01 is responsible for generating the strobe signal that latches data into the digital-to-analog converter. The clocked-serial input can be characterized as a sequence of bursts of activity interspersed with periods of quiescence. Internally CSB01 is a counter that is reset at the beginning of each burst and counts up the number of clock cycles that are seen, finally asserting its output briefly when all twelve data bits have been accumulated by CSA01. This output is then used as a strobe for the parallel data. Thus, given a sequence of incoming data words, CSB01 asserts its output once per word. The behavior of CSB01 is described in terms of frequencies; the output frequency is positive only when the input frequency is positive.

CSB01 is a good example of how explicit knowledge about functional organization simplifies troubleshooting. Simulating the behaviors of the individual components — the two counters, the gates, and the pullup resistors — would be relatively tedious. Encapsulating them along with the feedback signal yields an aggregate behavior that is almost as easy to describe and simulate as that of just one counter. Furthermore, it lends itself to description in terms of frequencies and rates of change, features that are easier to observe than the individual counting steps.

3.2.2 Physical Organization of the Audio Decoder

The Audio Decoder is implemented using nine chips and a nine-resistor network that is treated by the program as an ordinary chip. The correspondence between the functional components and the physical chips is shown in Figure 3.7. The serial ser01 signal is carried by a clock and a data signal (n56 and n260, respectively). The parallel signal par01 corresponds to two control signals n290 and n232 and twelve bits of data, named (from most to least significant) n48, n289, n246, n88, n208, n139, n131, n112, n194, n159, n117 and n236.

The likelihood of failure for each chip is estimated from its physical complexity as measured by the count of pins. The probability that chip is normal is simply the probability that all its pins are normal. Wires are assumed not to fail.

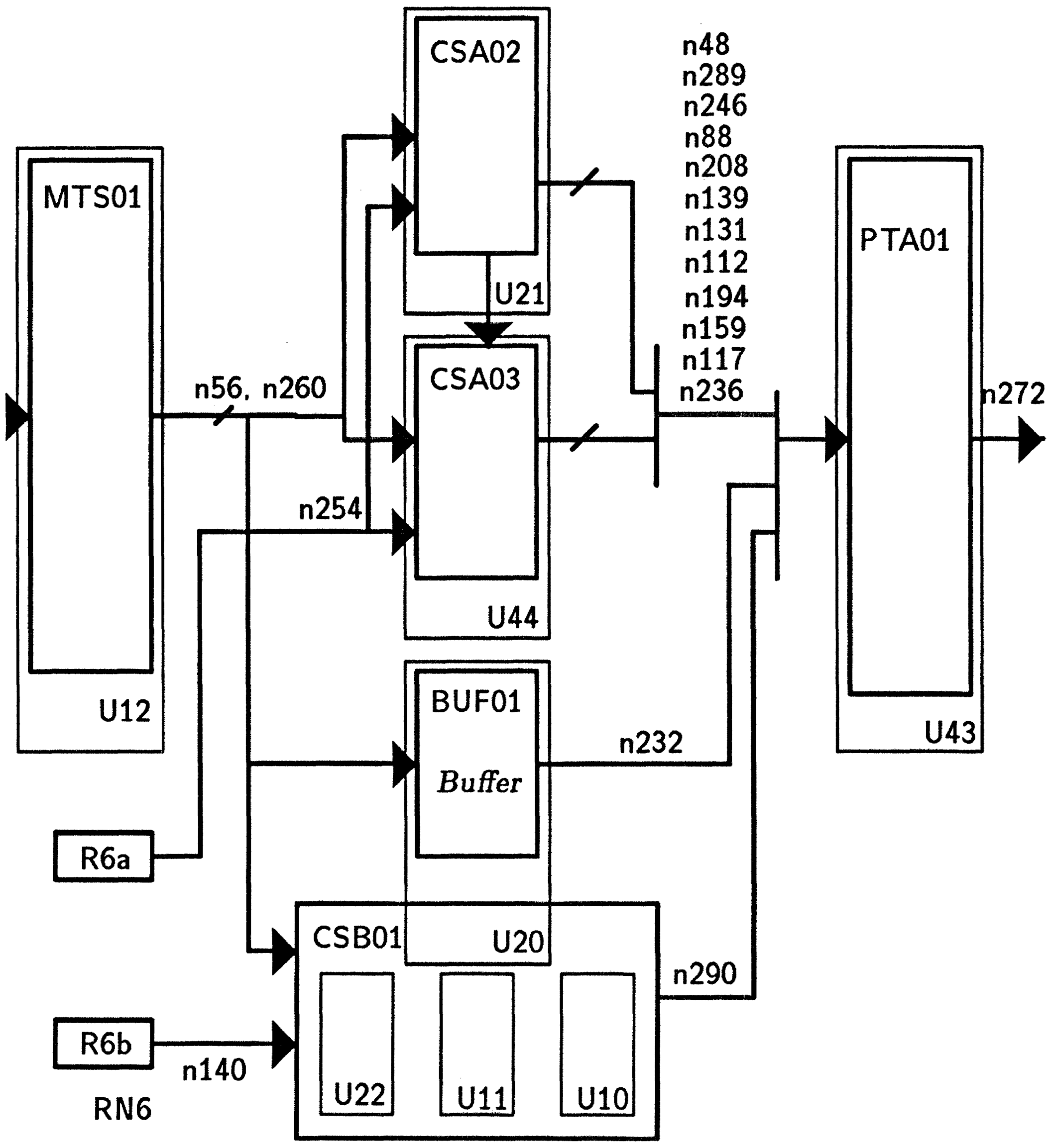
There are 130 pins in the audio circuit; in principle the program can suggest probing any of them. However, since etches are assumed not to fail, there is no need to probe more than one pin attached to any given etch, nor is there any need to probe pins that are attached directly to power or ground. Hence in this example there are only 23 distinguishable probes that XDE will ever suggest.

3.2.3 Audio Decoder Example I

Suppose that the output of PTA01 is observed to be flat, that is, zero frequency and amplitude. Any of the ten chips could be responsible, so there are ten singleton candidates, one corresponding to each chip. The candidate [U43] (the digital-to-analog converter) is judged to be somewhat likelier than the others.

The model makes predictions about which of the signals in the circuit should have a constant 1 value (n140, for example), which should have a constant 0 value (n34, for example, which is 0 except during certain local keyboard operations), and which should be changing. The program suggests a number of signals that could be examined, shown below:

Figure 3.7: Audio Decoder Physical Organization



Place	Expected	Entropy	Supporting Environments
n290	changing	.83	{RN6,U12,U10,U11,U20}
n280	changing	.76	{RN6,U12,U10,U20}
n112	changing	.73	{RN6,U12,U21,U44}
n88	changing	.60	{RN6,U12,U21}
...	...	...	...

The highest ranked probe is of n290, one of the write strobes to PTA01. This makes sense, since if this signal were dead it would explain why the output was flat and would tend to exonerate the shift registers that accumulate the incoming data bits. Suppose n290 is observed to be a constant 1 instead of changing as expected. Since it was supposed to be changing as long as the components {RN6, U12, U10, U11, U20} are all working properly, ⟨RN6, U12, U10, U11, U20⟩ is a conflict. There are now five candidates, one corresponding to each chip. [U12] is slightly likelier than the other candidates.

Now a different group of probes are ranked the highest. All of the signals on the data bus are equally good; if both of the candidates [RN6] and [U12] are working then these signals should be changing:

Place	Expected	Entropy	Supporting Environments
n88	changing	.72	{RN6,U12,U21}
n112	changing	.72	{RN6,U12,U21,U44}
n48	changing	.72	{RN6,U12,U21}
n159	changing	.72	{RN6,U12,U21,U44}
...	...	...	...

Suppose the data bit n88 is observed to have the constant value 1 instead of changing. Now ⟨RN6,U12,U21⟩ is a new conflict. There are two singleton candidates, [RN6] and [U12], and three two-component candidates: [U10, U21], [U11, U21] and [U20, U21]. The singleton candidates are judged to be the most likely, and U12 more likely to fail than RN6. The probes given the highest ranking are those of signals that are expected to be changing independently of whether RN6 is working or not, namely n56 and n232:

Place	Expected	Entropy	Supporting Environments
n56	changing	.92	{U12}
n232	changing	.92	{U12,U22}
...	...	...	...

Signal n56 is observed to have the constant value 1, so ⟨U12⟩ is a conflict and hence U12 is the only singleton candidate.

The troubleshooting program performs well on this example; only three probes in addition to the initial symptom were needed to yield a single candidate with much higher probability than the others (transcript in Appendix A.2). More important, it was able to do so using only temporally coarse predictions about the signals in the circuit, predictions in terms that corresponded directly to probes that the troubleshooter could make easily.

3.2.4 Audio Decoder Example II

Troubleshooting a second example with the same initial symptoms but a different underlying fault yields poorer performance. By including information about the way components are expected to fail, however, its performance improves dramatically (transcripts in Appendices A.4 and A.5).

Initially the output is observed to be flat and instead of changing as expected, n290 is observed to be constant 1. As before, the five most likely candidates are [RN6], [U12], [U10], [U11], and [U20]. This time, however, n88 is observed to be changing, as would be expected if everything were normal. Given no change in the set of candidates, probes of other data bus bits still appear to be the most informative probes; for example, the next set of suggestions is shown below:

Place	Expected	Entropy	Supporting Environments
n236	changing	.72	{RN6,U12,U21,U44}
n208	changing	.72	{RN6,U12,U21}
n117	changing	.72	{RN6,U12,U21,U44}
n289	changing	.72	{RN6,U12,U21}
...	...	...	...

The next six probes similarly yield no new conflicts and do not change the set of candidates. Finally, the program suggests probing signal n213, the signal that was immediately upstream of the discrepancy observed at n290. If U20 is working properly, then it should be a constant 0. It is observed to be changing, hence ⟨U20⟩ is a conflict and [U20] the single highest ranked candidate.

The difficulty is that the program just did eight probes, six of which were useless. The fact that even one of the bits of the data bus was changing should have indicated that the problem was unlikely to be in U12 or RN6. This is because if either of those components were broken the entire bus would probably be inactive. Hence, the more informative probes would have been in the vicinity of CSB01.

Including fault models for the components in MTS01 and CSB01 changes the efficiency of the troubleshooter dramatically. Now, instead of suggesting probes of the data bus bits, the higher ranked probes are those around CSB01, the component responsible for producing the discrepant signal n290. In particular, n213 is now among the highest ranked probes:

Place	Expected	Entropy	Supporting Environments
n213	changing 0	1.0	{RN6,U10,U11,U12,U20} {U20}
n56	changing	1.0	{U12}
n159	changing	0.79	{RN6,U12,U21,U44}
n289	changing	0.79	{RN6,U12,U21}
...	...	...	...

When n213 is observed to be changing, the ⟨U20⟩ is a conflict and so [U20] becomes the single likeliest candidate. Instead of making eight probes, this time the program only makes two. Furthermore, using fault models as heuristics does not decrease the performance of the other troubleshooting examples. The other scenarios shown require the same number of probes with or without fault models.

3.2.5 Audio Decoder Example III

Suppose that instead of the output being simply flat, its amplitude and frequency are correct, but it is distorted as was shown earlier in Figure 3.6 (Page 47). Using only temporally coarse descriptions of signals, the troubleshooting program is able to isolate the responsible component using sixteen probes.

The initial symptom is that the frequency of zero crossings in the first derivative of the output signal is higher than expected. All components are singleton candidates, and as in previous examples the first probe is at the write strobe signal n290. This signal is expected to be changing, and it is. The next two probes are at internal signals of CSB01, and appear to be changing as expected.

All but one of the next eleven probes are of the data bus bits, which are all expected to be changing. The temporally coarse behavior model does not include enough detail to indicate which of the bits ought to be probed first; any of the twelve bits having the wrong value at the moment they are latched into PTA01 could result in a distortion similar to that described. Eventually the signal n246 is discovered to be stuck at 1, yielding the conflict ⟨RN6,U21⟩ and hence the likeliest candidates as [U21] and [RN6]. After two more probes the conflict ⟨U21⟩ is discovered and the candidate [U21] is left as the final diagnosis (transcript in Appendix A.7).

What is interesting about the performance of the troubleshooting program using the temporally coarse model is not the probes it did, but the probes it did not do. The serial signals n56 and n260, for example, were not probed, and this makes sense: if there were faults there or upstream of there, the effects would probably have been more drastic than mere distortion of the output. Sixteen probes may seem like a lot, but it would require a considerably more temporally detailed (and expensive) model to do much better. To determine without probing exactly which data bus bits were wrong, for example, would have required being able to observe the shape of the output to twelve bits of precision and at just those moments when the write strobe signals were asserted. While this is not impossible, human troubleshooters rarely go to that kind of trouble, preferring instead to do a few more simple probes.

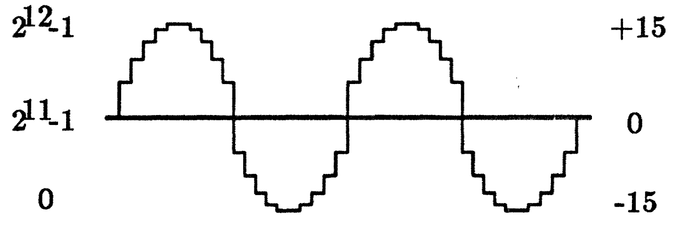
3.2.6 Audio Decoder Example IV

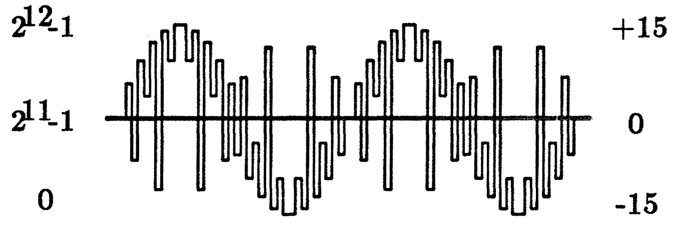
Like any abstraction, temporally coarse behavior models discard information. The final Audio Decoder troubleshooting example is similar to the previous example, illustrating that temporally coarse models discard information that could potentially have been used to improve the choice of probes.

The initial symptom is that the amplitude of the audio output signal is correct, but the frequency of its zero crossings is much higher than expected. Figure 3.8 shows the expected signal and that observed.

The initial probe of the write strobe n290 reveals that the signal is changing, just as expected. The subsequent probe of n280, a signal inside CSB01, however, reveals that it is a constant 1 instead of changing, as expected. This produces the conflict ⟨RN6, U12, U22, U10, U11⟩, and those five components are the top candidates.

Figure 3.8: Signal with Too Many Zero Crossings



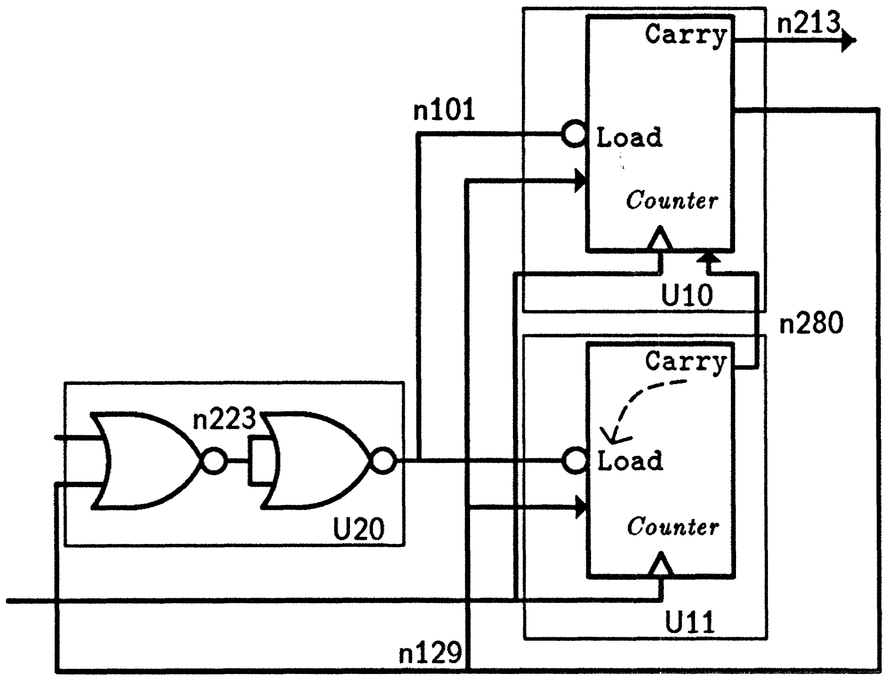


The observation that n280 is 1 triggers some new predictions (dashed arrow in Figure 3.9). n280 is the carry-out signal of the four-bit counter U11. Since that output is 0 whenever the Load control input to the counter is 0, the model concludes that if the counter is working normally, then the load control input n101 must have been a constant 1.

After seven probes elsewhere in the circuit, the program suggests probing n101. It is observed to be changing, hence the counter U11 cannot be working normally. Hence ⟨U11⟩ is a singleton conflict and [U11] is the single highest ranked candidate.

The program reached a diagnosis with eleven probes. As in the previous example, this may seem like a lot, but it would require a much more temporally detailed model to do better. For example, one of these probes is of the data bus signal n289, which was predicted to be changing. But there is no distortion of the data signals that could account for the observed distortion: the basic problem is that the rate at which the output signal is changing is higher than expected — the data values are getting strobed too fast. This can only be caused by a clock running too fast or some defect in the burst detection counters. In fact that is just what is happening: the

Figure 3.9: Internal Structure of CSB01



carry-out of the low order counter bits is stuck, so CSB01 asserts its output twice as fast as expected. Half of these times the data is correct (which is why the output is still recognizably sinusoidal) but the other times it is just transient garbage from the accumulator CSA01. Probes of the data bus signals in this example are not as likely to be informative as probes in the circuitry responsible for generating the write strobes. Any such inferences, however, depend upon being able to observe and reason about the details of the output waveform and the moment-to-moment activity of the clock and data signals, which would be sufficiently expensive that a few extra probes are an acceptable alternative.

3.2.7 Summary of the Audio Decoder Examples

The Audio Decoder circuit used in these troubleshooting examples illustrates the effectiveness and limitations of temporally abstract models of circuit behavior. The functional organization used in the model explicitly represents relationships between the rates of change on the inputs and outputs of components. These signal features are easy for the troubleshooter to observe, and so define an appropriate vocabulary with which to describe their behavior. These temporally coarse behavior descriptions are associated with the functional organization of the circuit. For example, the three chips U10, U11, and U22 not only have their own behavior descriptions, but there is a temporally coarse description of CSB01, the composition of all three. The temporally coarse descriptions are adequate for troubleshooting many of the possible failures, although there are cases for which a more temporally detailed model would provide the same diagnoses with fewer probes, and others for which the temporally coarse observations and models cannot provide a unique diagnosis no matter how many probes are done.

3.3 Input Encoder Examples

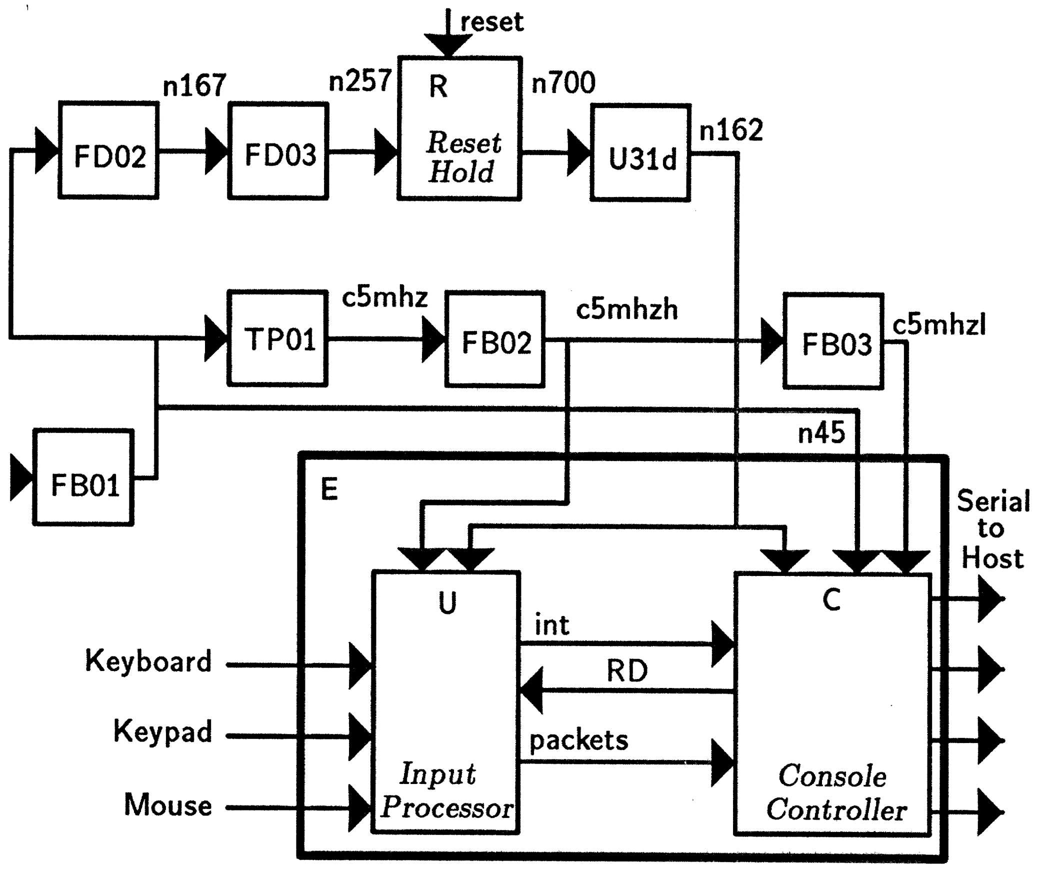
The purpose of the Console Controller Board is to transmit keystrokes — both up- and down- transitions — and mouse activity to the 3600 host computer. In addition, certain keystroke sequences starting with a down transition on the "local" key cause changes in local display parameters, such as the brightness of the screen. The section of the board responsible for these activities is the Input Encoder. In the following sections the structure and behavior of the Input Encoder will be presented in more detail than the simplified view given in the Chapter 1. The troubleshooting examples that involve it illustrate how temporal abstractions drastically simplify reasoning about devices with sequential feedback and internal state, so much so that model-based troubleshooting can apply to board-scale digital circuits.

3.3.1 Functional Organization of the Input Encoder

The Input Encoder merges three streams of data from the console peripherals and encodes them in packets to be sent the host (Figure 3.10). The three information streams are:

- Each up- and down-transition on the keys of the main keyboard is encoded as a single packet.
- An auxiliary numeric keypad with fewer keys than the main keyboard can be attached that produces up- and down-transitions, also encoded as single packets.
- Each change of mouse position or position of its three buttons causes a packet to be sent to the host.

Figure 3.10: Input Encoder Functional Organization



Transmission of packets is accomplished by the Input Processor (denoted U in Figure 3.10), which polls the keyboard, keypad, and mouse, asserting its interrupt line (int) whenever a key transition or mouse motion has occurred. When the int signal is asserted, the Console Controller C will respond by asserting the read signal RD a few instructions later. If the interrupt response time from the Console Controller is small enough (a few microseconds), a packet is correctly transmitted from U to C.

The Console Controller C interprets some keystroke packet sequences as local commands; for example, the sequence "Local key down, B key down, B key up, Local key up" will increase the brightness of the console screen. Other incoming packets are sent on to the host.

In addition to the power and ground inputs (not shown), the Input Processor and Console Controller both require a two-phase 5 Mhz clock signal, denoted c5mhz in Figure 3.10. These clocks are produced by the Clock Generator section described earlier. The components involved in generating and buffering the clocks are similar to those encountered earlier:

- The two phase clock generator TP01 converts a single-phase clock signal into two clock signals 180 degrees mutually out of phase.
- The frequency dividers FD02 and FD03 convert an incoming signal with frequency n into one with frequency $\frac{n}{2}$.
- The frequency buffers FB01, FB02, and FB03 produce output signals with the same frequency as their inputs.

Finally, both U and C also have an active-low "reset" input. When the reset button signal is asserted and the clock signal n257 is running, the Reset Hold Counter (denoted R) asserts n700 for at least 100ms, which initializes both the Input Processor and Console Controller.

3.3.2 Physical Organization of the Input Encoder

The Input Encoder implementation centers around an Intel 8035 microprocessor [Intel86]. Communication with the mouse and keyboard are done through a dedicated Intel 8741 microprocessor with onboard erasable program memory. The functional subcomponents of the Input Encoder are each implemented by one or more chips as shown in Figure 3.11 and Figure 3.12:

Figure 3.11: Input Encoder Schematic

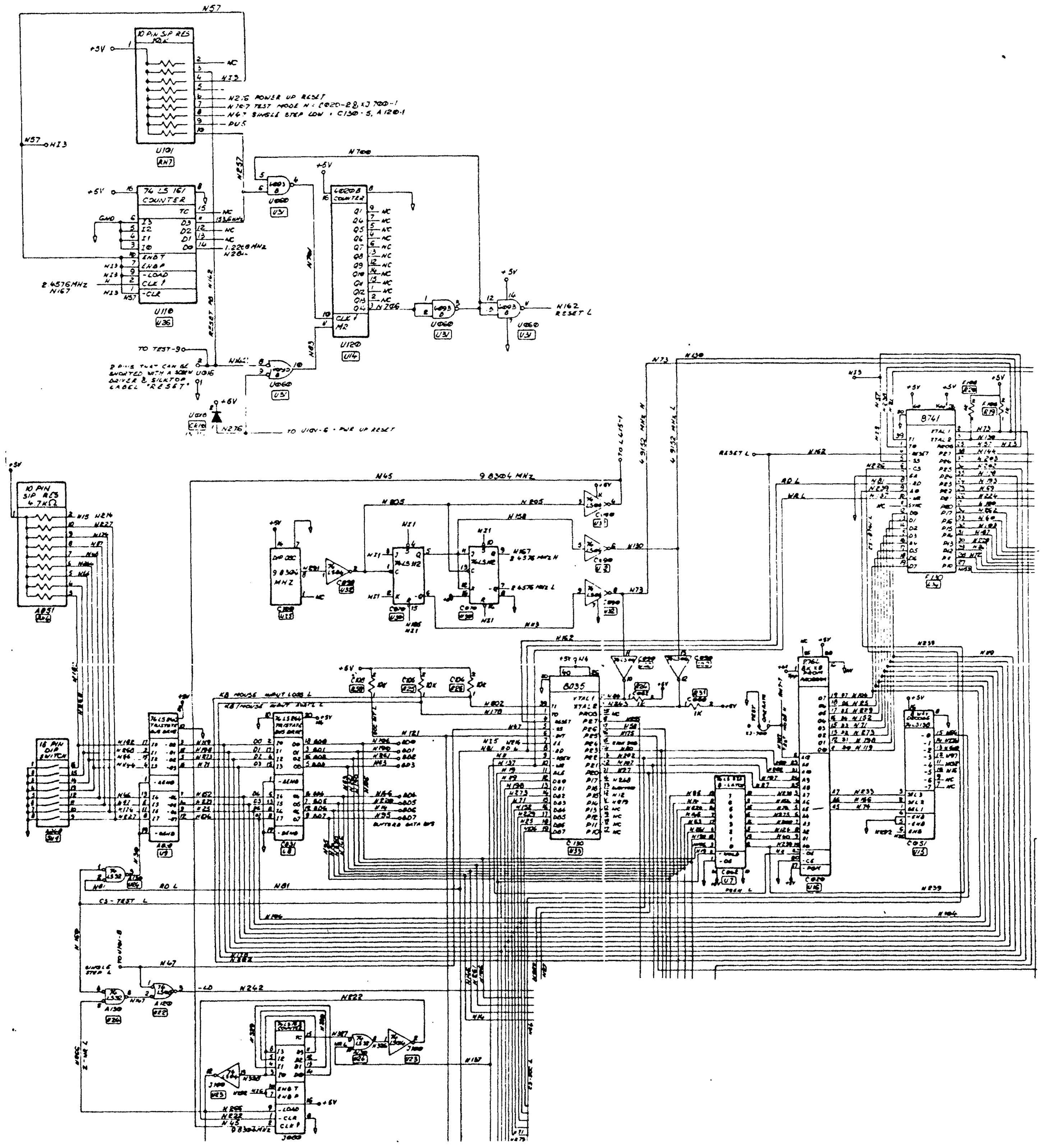
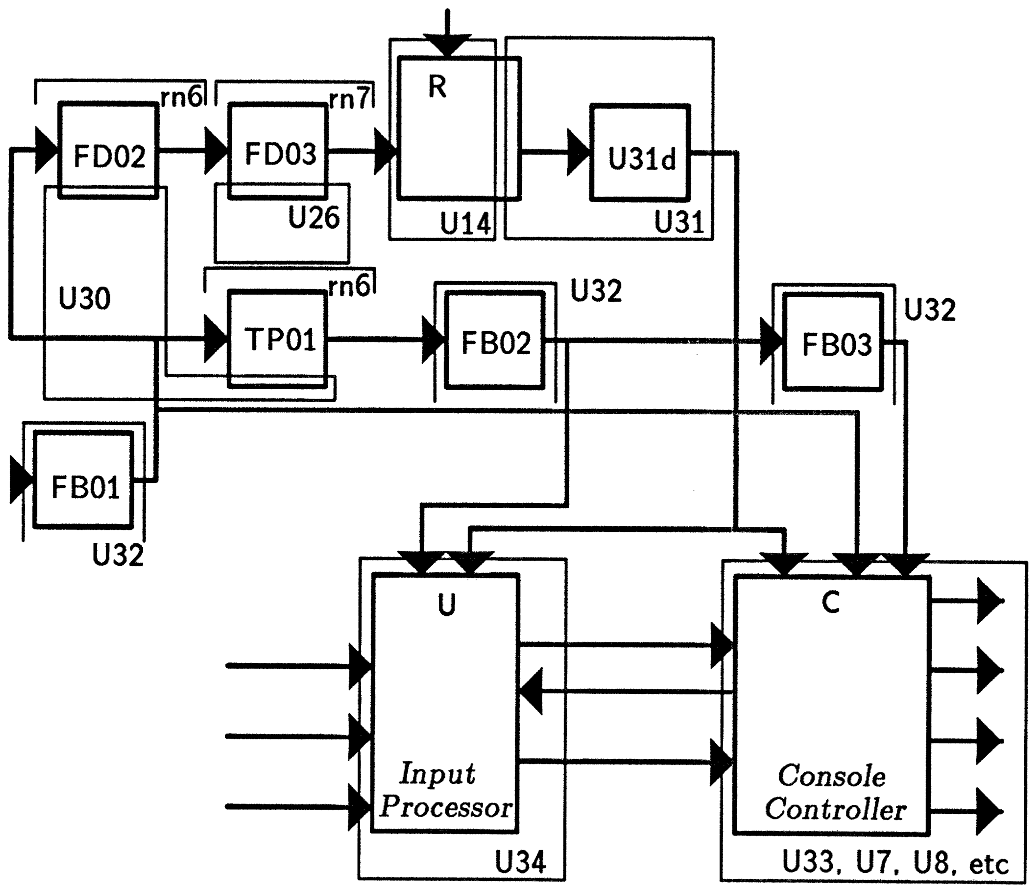


Figure 3.12: Input Encoder Physical Organization



- The Input Processor is implemented by the Intel 8741 microprocessor chip U34.
- The Console Controller is implemented by the physical 8035 processor U33 along with its external PROM (U18) and the buffers for its external bus data and control signals, involving chips U7, U8, U9, U13, and U24.
- The Reset Hold Counter is implemented by the 14-bit counter chip U14 and some NAND gates on chip U31.

- The remaining functions are implemented by the inverters and JK flipflops on chips U32 and U30.

3.3.3 Expected Behavior of the Input Encoder

A simple test of the Input Encoder consists of pressing and releasing the Reset button, then rolling the mouse around. The expected behavior of the Input Encoder is as follows:

- While the reset button is pressed the output of the Reset Hold Counter is held low. With the clock input n257 running at 153Khz, the signal goes high 100ms after releasing the button.
- The low-to-high transition on n162 causes the Input Processor – with its clock input running at 5 Mhz – to go from the "stop" state, to the "run" state in which it transmits keyboard and mouse transitions to the Console Controller.
- The low-to-high transition on n162 causes the Console Controller – with its clock input running at 5 Mhz – to go from the "reset" state to the "init" state and then to the "monitor" state in which it responds to interrupts and transmits incoming packets to the host.
- Each $\frac{1}{100}$ inch of mouse motion causes the Input Processor to interrupt the Console Controller, and because the Console Controller is in the "monitor" state it is responding to interrupts and a mouse position update is sent to the Console Controller.
- The Console Controller sends the mouse position update to the host.

Each of these high level behaviors has implications for the activity of certain observable signals. The important ones for this example are:

- The reset signal n700 will be low, then go high. Vice versa the signal n162, since u31d is an inverter.
- The active-low interrupt output of U will stay high while the mouse is still, and will be rapidly asserted and deasserted while the mouse moves.

- The select signal for the 8741 (n226) will remain high except for a short time after U interrupts. While in any state other than the "reset" state the read signal RD (n81) and other bus signals will have frequencies dependent on the input clock rate of Console Controller.

3.3.4 Finding a faulty Input Processor

Suppose that upon rolling the mouse around, the mouse cursor at the host does not move. This is recorded as a discrepancy at the output of E. The model predicts that the transition should have been sent if all sixteen chips were working, but since it was not, the conflict ⟨U7, U8, U9, U13, U14, U15, U16, U22, U23, U24, U25, U30, U31, U32, U33, U34⟩ results. There are sixteen candidates above threshold, the top few of which are shown below (transcript in Appendix A.10). The notation $\mathrm{U25}_{\mathbf{Open}}$ means that one of the known fault modes for U25, called "Open," is consistent with the observations so far; [$\mathrm{U25}_{\mathbf{Open}}$] comes out on top because that failure of U25 is likelier than other any other chip failure, as discussed in the Clock Generator scenario:

Weight	Candidate	Note
0.135	[$\mathrm{U25}_{\mathbf{Open}}$]	Oscillator chip
0.102	[U33]	8035 Microprocessor
0.102	[U34]	8741 Microprocessor
0.072	[U16]	PROM
...	...	

Among many predictions made by the model, the following ones are about observable signals. In this example, the frequencies of single- and two-phase clocks are taken to be observable.

Node	Signal	Expected	Support
n178	Interrupt	changing	{U25,U32,U30,U26, U31,U14,RN6,RN7}
n226	**U** Select	changing	{U25,U32,U30,U26,U31, U14,U31,U33,RN6,RN7}
n162	Reset	hi-lo-hi	{U25,U32,U30,U26,U31, U14,RN6,RN7}
n700	$\overline{\text{Reset}}$	lo-hi-lo	{U25,U32,U30,U26,U31, U14,RN6,RN7}
n137	Write	27 Khz	{U25,U32,U30,U25,U31, U14,RN6,RN7,U33}
n257	153Khz	153 Khz	{U25,U32,U30,U26,RN6,RN7}
c5mhz	Clock	5 Mhz	{U25,U32,U30,RN6,RN7}
c5mhzl	**U** Clock	5 Mhz	{U25,U32,U30,RN6,RN7}
c5mhzh	**C** Clock	5 Mhz	{U25,U32,U30,RN6,RN7}
...	...	...	...

XDE suggests **n178**, the interrupt line, as the most informative probe – more specifically, it suggests that the signal be probed to see whether it changes while the mouse is being moved.

This probe selection is the single most interesting inference in this example, and it is important to understand why it was made. In a purely mechanistic sense, XDE suggested the interrupt line because if a discrepancy were observed there, the conflict ⟨U25, U32, U30, U26, U31, U14, U31, RN6, RN7⟩ would result, thereby (approximately) halving the candidate set. From a modeling point of view, the interesting point is that a crude, temporally abstract model of the behavior of the Input Processor is adequate to infer that so long as **U** is working properly, has power and clock inputs, and is not being reset, that motions of the mouse will activate the interrupt line. Similarly, if keys were being pressed, again the interrupt line would be active. Abstracting the 8741 microprocessor to a two-state device makes prediction of its behavior in this example much simpler than doing instruction-level simulation, and still provides predictions that are diagnostically useful.

Returning to the example, suppose **n178** is observed to be a constant **1**. This yields ⟨U25, U32, U30, U26, U31, U14, U31, RN6, RN7⟩ as a conflict, and the top four candidates are as shown below. [$\text{U25}_{\textbf{Open}}$] comes out on top as before:

Weight	Candidate	Note
0.280	[$U25_{Open}$]	Oscillator chip
0.212	[U34]	8741 Microprocessor
0.085	[U14]	14-bit counter in Reset Hold Counter
0.085	[U26]	4-bit counter in FD03
...	...	...

This yields a new set of predictions from among which XDE will select the next probe.

Node	Signal	Expected	Support
n162	Reset	hi-lo-hi	{U25,U32,U30,U26,U31,U14,RN6,RN7}
		lo	{U34,U25,U32,U30,RN6,RN7}
n700	$\overline{\text{Reset}}$	lo-hi-lo	{U25,U32,U30,U26,U31,U14,RN6,RN7}
		hi	{U34,U25,U32,U30,RN6,RN7}
n257	153Khz	153 Khz	{U25,U32,U30,U26,RN6,RN7}
		0 Hz	{$U25_{Open}$,U26,U30,U32,RN6,RN7}
c5mhz	Clock	5 Mhz	{U25,U32,U30,RN6,RN7}
		0 Hz	{$U25_{Open}$,U32,U30,RN6,RN7}
c5mhzl	U Clock	5 Mhz	{U25,U32,U30,RN6,RN7}
		0 Hz	{$U25_{Open}$,U32,U30,RN6,RN7}
c5mhzh	C Clock	5 Mhz	{U25,U32,U30,RN6,RN7}
		0 Hz	{$U25_{Open}$,U32,U30,RN6,RN7}
...	...	...	...

Note that node n162 now has two conflicting predictions for its behavior — the normal behavior, and the misbehavior that it is low at moments when it was expected to be high. The argument for the latter behavior is as follows. If U34 is working properly, U has a clock and incoming mouse motions. But since the int output was not asserted, then it must have been because U was in the "reset" state. Hence the reset input n162 must be asserted (low). This is the second most interesting inference in this example, and again, it is effective because the Input Processor has been reduced to a two-state device: only when the component models are so simple is it reasonable for the system to make inferences about component inputs from knowing their outputs.

The highest ranked probe is the input clock to Reset Hold Counter, n257, which is expected to have a frequency of 153 Khz if {U25, U32, U30, U26, RN6, RN7} are all working. Probing this signal, it is discovered to have the correct frequency.

This observation has two major consequences. First, it makes the likeliest candidate [$U25_{Open}$] inconsistent. Second, although it makes no new predictions, it does add new support to some predictions already present. For example, it was already believed that n167 had frequency 2.5 Mhz if {U25, U30, RN6, RN7} were working; it can now be deduced that it has frequency 2.5 Mhz if {U26, RN6, RN7} are working. Similarly c5mhz has frequency 5 Mhz if {U26, U30, RN6, RN7} are working, and so on. These inferences result in a new conflict, ⟨U26, U30, U32, U34, RN6, RN7⟩, so that the resulting highest ranked candidates are:

Weight	Candidate	Note
0.332	[U34]	8741 Microprocessor
0.132	[U30]	Frequency dividers
0.132	[U14]	Gates in Reset Hold Counter
0.116	[U32]	Frequency buffers
0.116	[U31]	Counter in Reset Hold Counter
0.083	[RN6]	Pullups
0.083	[RN7]	Pullups

The reset signal n162 is now the highest ranked probe. Probing it shows that it is behaving normally – it starts out high, then goes low while the reset button is pressed, then returns high a short time later. Our observation technology is sufficiently crude that it is impossible to say exactly *when* the line went low – the essential observations are that (i) it was asserted long enough to reset U and C, and (ii) it was unasserted while the mouse was rolling around. Nevertheless, for simplicity, the observation that gets recorded is that n700 was high and low at just the times expected. There are now just five candidates:

Weight	Candidate	Note
0.449	[U34]	8741 Microprocessor
0.184	[U30]	Frequency dividers
0.162	[U32]	Frequency buffers
0.118	[RN6]	Pullups
0.118	[RN7]	Pullups

Next, an observation is suggested at c5mhz. Doing so reveals that it has the expected frequency of 5 Mhz. After several more corroborating probes of clock signals, new conflicts are discovered and candidates eliminated. Eventually the only remaining candidates are:

Weight	Candidate	Note
0.800	[U34]	8741 Microprocessor
0.200	[RN7]	Pullups

A final corroborating probe at node n57 (not shown in Figure 3.10) results in the sole candidate:

Weight	Candidate	Note
1.000	[U34]	8741 Microprocessor

This example is the same as that presented in Chapter 1 and has the same moral: what is interesting about it is the contrast between the simplicity of the reasoning and the relatively few probes (eleven, to be exact) required to isolate the fault to a single chip, in spite of the underlying complexity of the circuit. What made that simplicity possible was the choice of behavioral abstractions, in particular the temporally coarse behavior models for the microprocessors, which made it possible to reason about the reset and interrupt signals without getting swamped in details.

3.3.5 Finding a faulty Console Controller

The preceding example illustrates the important characteristics of the behavior models for the Input Encoder examples. Another example illustrates how the program isolates faults inside the functional component C (transcript in Appendix A.11).

The initial inputs and symptoms are the same as before, so the interrupt signal n178 is suggested as the next probe point. This time, it is observed to be changing while the mouse is rolled around. This suggests that it is not the Input Processor U that is working normally. Probing the clock signal n257 shows that its frequency is normal, suggesting that the Clock Generator section is working normally as well. This leaves twelve candidates, the top five of which are inside the Console Controller C:

Weight	Candidate	Note
0.165	[U33]	8035 Microprocessor
0.115	[U16]	PROM
0.082	[U7]	Instruction Address Latch
0.082	[U9]	Buffer
0.082	[U8]	Buffer
...	...	

The behavior of the 8035 microprocessor inside E is described in a temporally coarse fashion, just as the 8741 microprocessor was in the previous example. The 8035 is either in the "run" or "stop" state, depending on the frequency of its clock input and whether its reset input is asserted. While running, it should be repeatedly asserting the signal PSEN, which reads instructions from the PROM. If the PROM and some other buffers are all working properly, then the Read and Write bus control signal should be repeatedly asserted as well. The top ranked probes are shown here:

Node	Signal	Expected	Support
n81	Read	changing	{U7,U8,U9,U30,U32,U33,RN6}
n137	Write	changing	{U7,U8,U9,U30,U32,U33,RN6}
n11	PSEN	changing	{U30,U32,U33,RN6}
...	...	...	...

After observing that none of these three signals are changing, there are just four candidates:

Weight	Candidate	Note
0.488	[U33]	8035 Microprocessor
0.195	[U30]	Frequency dividers
0.171	[U32]	Frequency buffers
0.122	[RN6]	Pullups

Two subsequent probes of the clock inputs to the 8035 microprocessor U33 show that they are normal and leave U33 as the only candidate.

As before, the temporally coarse model of the behavior of the microprocessor and its combined behavior with the PROM and other components allowed a few simple probes (nine, in this example) to find the broken microprocessor.

3.4 Summary of Troubleshooting Scenarios

The seven scenarios presented above provide context and a set of examples that the next few chapters will draw upon. They also illustrate that the troubleshooting engine XDE is able to deal with complex devices not due to any major innovation in the underlying model-based troubleshooting technology, but rather due to innovations in constructing the device model that it uses. XDE works well on the Console Controller Board because the board can be modeled with the goal of troubleshooting explicitly in mind, and this implies certain desirable features of that model. Temporally coarse descriptions of behavior are obviously important, but there are others. The following three chapters will present in detail the representations of circuit structure, circuit behavior, and faults that all together can represent complex devices in a way that makes it feasible for XDE or any other model-based troubleshooting engine to troubleshoot them.

Chapter 4

Representing Circuit Structure

Model-based troubleshooting requires an explicit representation of the internal structure of the device being diagnosed. All the diagnoses that the troubleshooting engine produces will be expressed in terms of the components that appear in that structure representation. The need for efficiency indicates several desirable properties of this structure representation: it should be a strict hierarchy, its leaves should correspond to the locations of possible failures, and every field replaceable component should correspond to some node in the hierarchy. These properties are embodied in a representation of the *physical* structure of the device. Predicting the behavior of a complex device from the details of its physical organization can be greatly simplified by using a representation of the intended behaviors of groups of components at multiple levels of abstraction. For example, it is easier to reason about the behavior of a digital logic gate than about the equivalent collection of resistors and transistors: the structural composition of those components enables abstraction of their combined behavior. For the same reason, it is easier to reason about an adder performing arithmetic on integers than about the equivalent collection digital logic gates, and so on. A nonstrict *functional* hierarchy provides a way of organizing these structural compositions to which intended behaviors are attached. The nodes in the functional hierarchy are essentially "slices" through the physical structure [Sussman77] [Sussman80]. They are chosen explicitly to facilitate behavioral abstraction.

These two views of digital circuit organization are concretely expressed in the circuit structure language BASIL[1]. BASIL descends from DPL [Batali81] and TDL [Davis83] and it inherits the idea of representing circuit structures as graphs of objects with connections between them at "ports," although BASIL is implemented quite differently. BASIL provides predicates and a vocabulary of primitive components, but more important than BASIL itself are the principles for composing these primitives into physical and functional organizations in ways that facilitate troubleshooting. Two key principles are:

- Components in the representation of physical structure should correspond to the possible repairs.
- Structural composition should allow simplification of behaviors and facilitate behavioral abstraction.

4.1 Physical Organization

A representation of the internal physical organization of devices is essential in model-based troubleshooting. The physical world is where the observations that the troubleshooting engine requests and the repairs that it recommends are located; the physical world is also the source of information about the plausible failures.

4.1.1 Primitive Components

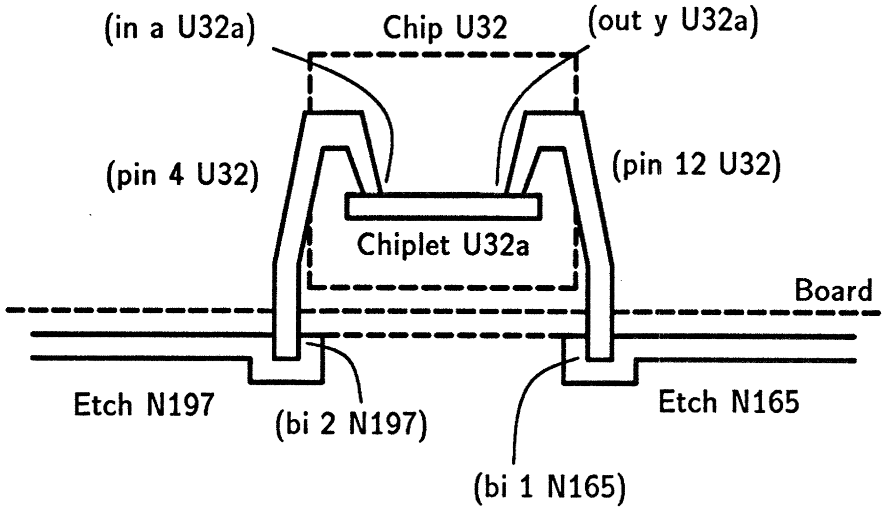
To represent the physical structure of a device for troubleshooting, the first and central issue is choosing the primitive level of detail. Since the complexity of the world is to be abstracted away into a graph of components and their connections, the essence of the choice is in where to draw those primitive component boundaries. Drawing these boundaries makes three fundamental commitments. First, it makes some failures indistinguishable to the troubleshooting engine — every failure inside a primitive component will result in the same diagnosis. Second, it makes some failures representable only as failures in multiple components — for example, the troubleshooting engine would diagnose a short circuit between two (supposedly) non-interacting components as failures in both components. Third, the lower the level of

[1]Box And String Interconnect Language.

physical detail the more work will be involved in predicting behavior — for example, representing individual transistors on a chip implies the possibility that the behavior of each individual transistor will be reasoned about explicitly. Thus there is a tradeoff to be made between the detail in the representation and the efficiency of reasoning with it: more detail makes diagnosis more accurate but results in more work. BASIL or any other structure representation is a compromise between these conflicting goals.

BASIL uses *etches*, *pins*, and *chiplets* (areas of silicon real estate inside the chip package) as its primitive components. Figure 4.1 shows a cross section of a chip soldered into a board. The etches, pins, and chiplet are all components. The principles at work in choosing these as primitives are discussed below.

Figure 4.1: Chip Cross Section



- Collect fault locations with indistinguishable effects into a single component.

Electrical signals travel between the etch and the silicon inside chips through a solder joint at the hole, the pin on the chip, and a tiny bonding wire that reaches from the pin to a metal pad on the silicon. Opens and shorts can happen to the pin proper, the bonding wires, and the solder; the bonding wire is especially susceptible to being shaken loose and becoming an open circuit. Under the assumption that only the voltages and currents at the solder joint will be observable, these physical failures are indistinguishable. Thus they are all treated as one component, called a *pin*. The pin has a port at each end, referred to here as the *solder* and the *bond* ports.

- Collect many individually unlikely fault locations into a single component.

The metal strips that run between the holes in a board are called etches. They are usually tree-structured when connecting more than two holes. Sometimes branches of the etches crack (becoming open circuits) or get accidentally connected to other etches (becoming short circuits or "bridging" faults). Such failures are somewhat less likely than the bonding-wire breaks mentioned above. BASIL thus represents an entire metal etch — no matter how many branches it has — as a single component with one port at each hole. There is no distinction between cracks in different branches of the etch; any real break will be diagnosed as a failure of the entire etch. There is also no representation of the physical adjacency of different etches and no way to explicitly represent bridging faults; real shorts between etches will be misdiagnosed as a pair of failures in the two etches.

BASIL could represent each branch and junction of the etch explicitly, and could represent the points of possible bridging explicitly, but this would entail an unreasonable number of primitive components. It would be inefficient since these faults are not nearly as common in the field as others. An alternative would be to represent the possible points of failure implicitly by representing the three-dimensional layout of the etches; this has not been done either.

The internal structure of chip packages provides another example of this principle. Every transistor on a silicon chip may produce a detectably different misbehavior if it fails, but any individual failure is relatively unlikely. Hence each independent functional unit on the silicon within a chip is a primitive component, referred to as a *chiplet*. For example, a 74LS04 chip has six inverters on it; each of these inverters is a separate chiplet within the chip.

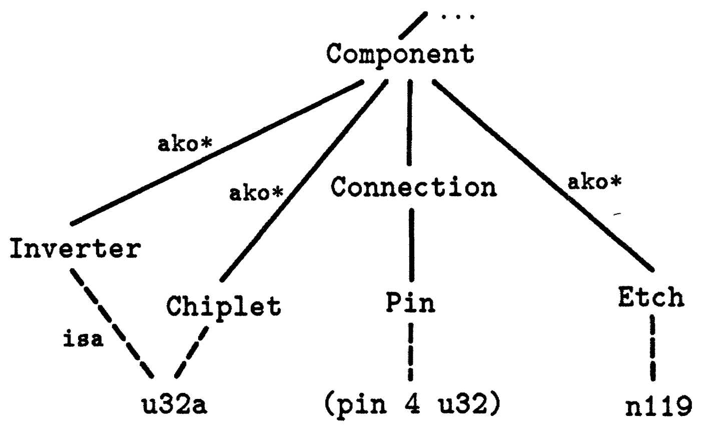
4.1.2 BASIL

BASIL represents the types of components and their relationships using four predicates. The basic syntax is Cambridge prefix predicate calculus using `[...]` to indicate predicate terms and `(...)` to indicate function terms. The syntax is inherited from JOSHUA [Rowley87].

The predicate **ako** forms the lattice of types. `[ako ?x ?y]` means that all individuals of type `?x` are of type `?y` also. For example, etches are a kind of component: `[ako etch component]`. The predicate **ako*** is the Kleene star of **ako**.

Among the primitive types of component are `etch`, `chiplet`, `pin`, `inverter`, `resistor`, and `switch`. Figure 4.2 shows a small portion of the type hierarchy.

Figure 4.2: Abbreviated AKO hierarchy



The predicate `isa` denotes the most specific types of an individual. For example, **u32a** is a physical realization of an inverter in silicon; hence it is both a `chiplet` and an `inverter`. These are denoted `[isa u32a chiplet]` and `[isa u32a inverter]`. The predicate `isa+` denotes the relationship between an individual and all of the types to which it belongs. Thus `[isa+ ?x ?z] == [isa ?x ?y]` $\wedge$ `[ako* ?y ?z]`. For example, u32a is a component because it is an inverter, and inverters are components: `[isa+ u32a component]`.

The chip cross-section in Figure 4.1 showed the following set of `isa` relations:

```
[isa n197 etch]  [isa (pin 4 u32) pin]   [isa u32a chiplet]
[isa n165 etch]  [isa (pin 12 u32) pin]  [isa u32a inverter]
```

Components interact with other components through *ports*. By convention a port denoted (?direction ?id ?component) is a port of that component. The direction function is one of in, out, or bi indicating that it is intended to be an input, output, or bidirectional port respectively. For example, (in a u32a) is the "a" input of inverter u32a. (bi 2 n119) is the port where etch n119 electrically interacts with pin 4 of U32. The predicate has-port denotes this relationship; for example, u32a has an "a" input: [has-port u32a (in a u32a)]. The set of assertions about ports shown in the chip cross-section of Figure 4.1 is:

```
[has-port n197 (bi 2 n197)]  [has-port n165 (bi 1 n165)]
[has-port u32a (in a u32a)]  [has-port u32a (out y u32a)]
```

Connections are a kind of component that have exactly two ports. Each of these ports is shared with one other component. The only kind of connection shown so far are pins, which are named (pin ?number ?chip)[2]. For example, in the chip cross-section of Figure 4.1, pin 4 of chip U32 connects port 2 of etch n119 to input port "a" of inverter u32a. This is denoted with the predicate conn as [conn (pin 4 u32) (bi 2 n119) (in a u32a)]. Note that in BASIL the only substantive difference between ordinary components and connections is that the names of the ports of a connection refer to adjacent components, not the connection itself. The connections shown in the chip cross-section of Figure 4.1 are the two pins:

```
[conn (pin 4 u32) (bi 2 n197) (in a u32)]
[conn (pin 12 u32) (bi 1 n165) (out y u32)]
```

BASIL has other predicates and a more densely populated ako hierarchy than indicated here. These details will be presented shortly.

[2]Most components are named by a single atom such as u32. Pins are the sole exception, since names like (pin 4 u32) are function terms. They could just as easily have been named by atoms, for example "u32.4."

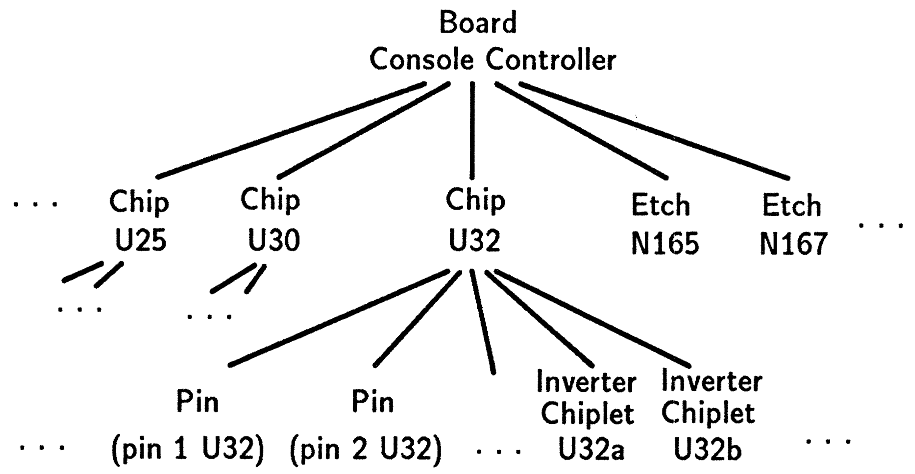
4.1.3 The Physical Part-Of Hierarchy

The predicates and primitive component types in BASIL allow an entire circuit board to be described in terms of the subparts of its chips and the connectivity among them, but it would be inefficient to troubleshoot a large circuit using only this primitive level of detail. Almost any symptom alone would yield dozens of pins, etches, and chiplets as suspects. A hierarchic representation allows groups of primitive components to be efficiently treated as a single component. For example, it is more efficient to diagnose to the level of chips before considering the internals of those chips, since there are far fewer chips to consider than pins and chiplets. The predicate `ppart-of` ("physical part of") denotes the relationship that forms the physical hierarchy; `[ppart-of u32a u32]` means that `u32a` is a part of `u32`.

The right physical components to group together to form the `ppart-of` hierarchy are the ones that correspond most directly to repair actions. The main objective of the troubleshooter is to find the repair or set of repairs most likely to make the device work again. Since the troubleshooting engine computes diagnoses that correspond to sets of components, it would be efficient to have a one-to-one correspondence between the possible repair actions and the components in the hierarchy. This would make each diagnosis map directly to a set of repairs to be done, and the troubleshooting engine would not waste effort distinguishing between different faults that had the same repair. In the circuit domain this is straightforward, since for the failures under consideration the possible repair actions consist only of replacing boards, replacing chips, and re-soldering broken etches. By making the hierarchy of components a physical hierarchy in which chips and boards are the only components other than the primitives, the diagnoses will be directly translatable into possible repairs. In the digital circuit domain the resulting hierarchy is bushy; one or more chiplets and their pins together form a `chip`, and chips and etches together form the `board`. Figure 4.3 shows a small portion of the physical part-of hierarchy of the Console Controller Board.

Manufactured artifacts can nearly always be decomposed into a part hierarchy that is strict, a decomposition that reflects the way the artifact was constructed. Chips are fabricated separately and soldered into the board, for example, and this indicates that the chips and printed board have no shared parts. There are exceptions whenever the assembly process itself causes boundaries to be diffuse. Parts may be built up by incremental and

Figure 4.3: A Portion of the **ppart-of** Relation



overlapping manufacturing steps, as with the layers of a silicon chip layout; parts may merge smoothly into one another, as with pieces of metal welded together. As long as the physical object can be divided along boundaries there is at least a degenerate strict hierarchy to be found: all of those parts can be immediate descendants of the overall structure. These exceptional and degenerate cases do not occur in digital circuit boards at the level of description that BASIL uses. Each pin and chiplet is part of just one chip, each bit of solder is part of one etch, and so on. The same would be true for larger scale organizations of boards, card cages, cabinets, and so on: the way the artifact gets assembled from its parts forms the physical part-of hierarchy. Even cables between different cabinets are not an exception; they customarily have their own part numbers and are typically listed in the parts list of the entire computer. The physical hierarchy in BASIL is strict and that accurately represents the real world.

The fact that the physical hierarchy is strict simplifies comparing alternative diagnoses. It need not be strict — the troubleshooting engine would still compare diagnoses and rank them appropriately — but a strict hierarchy makes it more efficient.

For troubleshooting, each component has a *status* indicating whether it is believed to be working normally, faulty in some known way, or faulty in an unknown way. The predicate `status-of` denotes this relation. For example, `[status-of u32 working]` means the component U32 is working, that is, it is not physically damaged. Because BASIL assumes that only components can fail, the status of each other component can be deduced from the status of the components that are part of it, or that it is part of. In the example above, the board is working if all its chips and etches are working; the chip U32 is working if all its pins and the six inverters inside it are working. Contrapositively, if U32 is not working then at least one of its pins or inverters is not working, and so on.

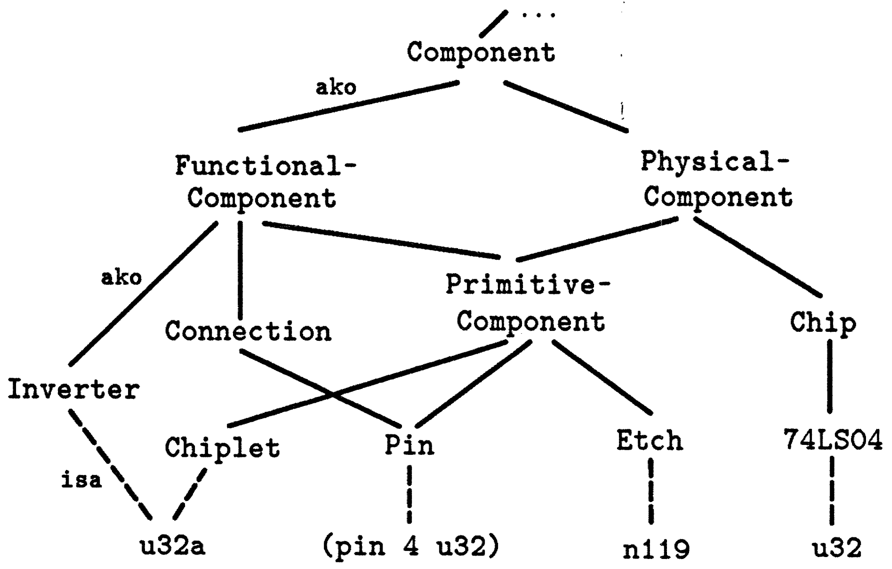
While troubleshooting, each status and each diagnosis is assigned a relative likelihood (as discussed in a later chapter). Computing these likelihoods is greatly simplified if the different component statuses can be treated as statistically independent. One way for that independence to be violated would be for components to share parts, since a single failure in some shared part would appear as a failure in all the sharing components (in fact, since the probabilities of failure in the two parent components would be different from their product, by definition their probabilities of failure are not independent). With a strict hierarchy it is trivial to determine whether parts are shared; a pair of components can share parts only if one is an ancestor of the other. The strict hierarchy thus simplifies computing relative likelihoods since it can easily be arranged that no diagnosis ever mentions failures in both a component and one of its ancestors.

4.2 Functional Organization

Although the physical organization discussed above is central to the troubleshooting task, the physical packaging of digital circuits often has an almost accidental nature. Implementing the desired functionality using off-the-shelf chips typically means sharing several functions in one package (for example, four gates on a chip) or using only a portion of the functions on a chip (for example, using a universal shift register with all its control inputs tied to power or ground). For efficient reasoning about the behavior of a complex device, it is useful to be able to consider the combined behavior of portions of several different physical components. Moreover, this reasoning requires behavioral abstractions, and some behavioral abstractions do not apply to primitive components. For example, it is simpler to reason about a digital adder operating on integers than about several one-bit adders doing their operations on bit vectors. Structural composition of those one-bit adders along with their interconnecting wires yield a composite component whose behavior can be described abstractly in terms of n-bit integers. That the one-bit adders reside on different physical components is an accident of implementation; together with their interconnecting wires they still form an adder component.

BASIL represents this knowledge as *functional* components augmenting the physical components described earlier. Functional components are similar in many ways to physical components; they have ports and statuses, and they are organized into a hierarchy by the `fpart-of` relation. The primitive components discussed earlier — etches, pins, and so on — are both physical and functional components; the functional and physical hierarchies thus meet at their leaves. This yields the expanded **ako** hierarchy shown in Figure 4.4.

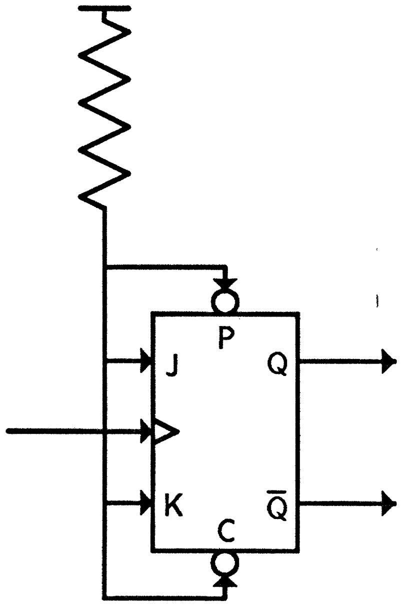
Figure 4.4: Expanded AKO Hierarchy



4.2.1 The Functional Part-Of Hierarchy

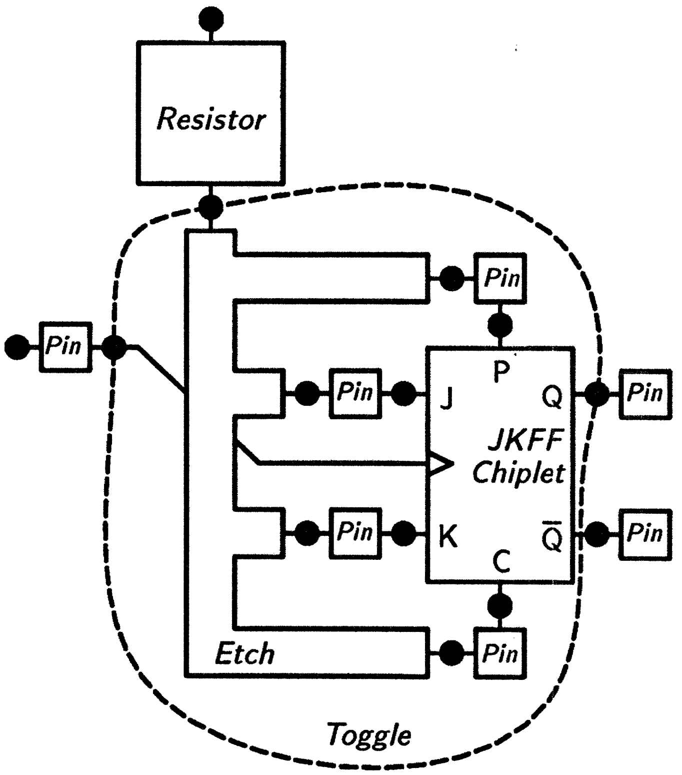
The functional part-of hierarchy is not strict, and has a much richer vocabulary of component types than the physical. The reason for this is that there are often several alternative and incomparable ways of describing even the same collection of components. For example, one way to describe the combination JK flipflop and pullup in Figure 4.5 is as a "Toggle," which has a one-bit output; another way would be as a "two-phase clock generator," which has a two-bit output. Both behavior descriptions are legitimate, but neither subsumes the other.

Figure 4.5: JK Flipflop Unencapsulated



The Toggle is as an example of a functional component that is the composition of several primitive components: (i) a JK flipflop chiplet (ii) the etch that connects four of its inputs together, and (iii) the pins that connect that etch to the chiplet. The etch, pins, and chiplet are all `fpart-of` the Toggle. Figure 4.6 shows these subcomponents (rectilinear boxes), the ports at which they interface (the black spots), and the boundaries of the Toggle (the dotted line). Both the JK flipflop and Toggle have an explicit "power" port that will be explained later; to avoid clutter these are not shown. Even with this simplification Figure 4.6 may be a bit difficult to understand; the difficulty of conveying encapsulations like this visually stems in part from the fact that etches and pins are not usually treated as explicit components, and from the fact that the desire to keep the boundary convex requires requires distortions of the normal two-dimensional layout.

Figure 4.6: JK Flipflop Encapsulated as a Toggle



The Toggle has ports just as the JK flipflop did. The relationship between the ports of an abstract component and the ports of its underlying components is represented with the predicate `corr` ("correspondence"):

- `[corr correspondence abstract-port . concrete-ports]` means that there is a group of one or more `concrete-ports` that correspond to one `abstract-port`. The nature of that correspondence is denoted by the `correspondence` argument. The most common correspondence is `identity`, which means that the two ports are equivalent. Other correspondences include `concat` (the concatenation of bits into integers), `ttl-power` (a high voltage port and a ground port that correspond to a single "power" port), and `two-phase-clock` (a pair of one-bit ports at which the voltages are 180 degrees out of phase).

In the case of the Toggle, each of its three ports stand in an `identity` correspondence with one underlying port apiece. These ports are indicated in Figure 4.6 where the dotted line passes through ports.

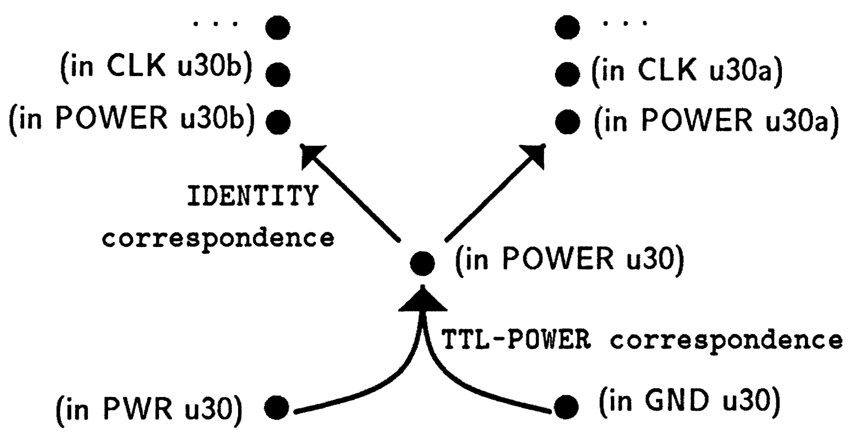
The `power` port that appears on nearly all chips introduces some complications, since the power supplied to all the chiplets on a typical chip is supplied through a pair of pins, "pwr" (high voltage) and "gnd." Figure 4.7 shows the `ttl-power` correspondence between the ports at these pins and a single `power` port shared by the whole chip. The `power` port for the whole chip then stands in an `identity` correspondence with each of the `power` ports of the chiplet components on the chip. U30, for example, is a dual JK flipflop chip, and its two flipflop chiplets are named U30a and U30b; they each have a `power` port along with their other ports (clk, for example). The advantage of the `power` port is that it somewhat simplifies the behavior descriptions of the individual components.

4.2.2 Principles for Structural Composition

Successive layers of possibly overlapping compositions can create a deep `fpart-of` lattice representing many behavioral groupings in the device. Yet it is one thing to be able to explicitly represent the hierarchic functional organization of a complex digital circuit, it is quite another to discover the right components to compose together and the right behavioral abstractions to use. Starting only from a digital circuit schematic and behaviorally detailed descriptions of the physical component behaviors, someone or something must construct that richer representation. Currently it is constructed by hand, but importantly, not in an *ad hoc* fashion. There is a fundamental principle at work:

- Structural composition should enable behavioral simplification.

Figure 4.7: Power Ports of Chip U30 and its Chiplets



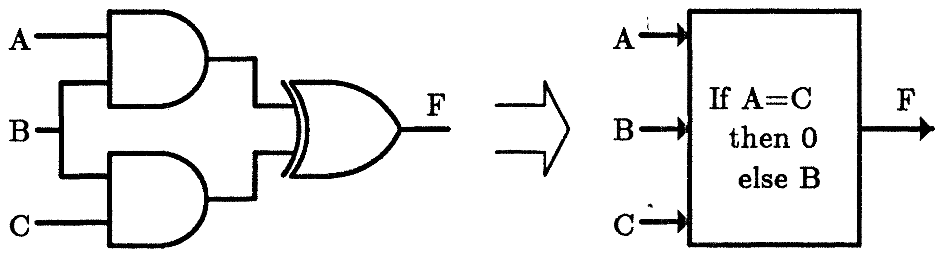
That is, the grouping of connected or related components together — structural composition — is distinct from behavioral abstraction, but from a troubleshooting perspective, the only motivation for structural composition is to simplify behavior. For example, there is no point in composing four one-bit adder slices together and calling it an "adder" unless the behavior associated with the adder takes advantage of the abstraction that maps from vectors of bits to integers. The Toggle is a worthwhile functional component because its behavior is much simpler than the whole JK flipflop. In digital circuits, there are three ways the general principle manifests itself and hence three reasons to introduce structural compositions:

1. To suppress constant signals. For example, if a node is pulled up and always supplies a "high" value to some component, the pullup and component can be grouped together to form a simpler component.

2. To encapsulate reconvergent signals. Reconvergent signals are signals that originate from a common source, and then are recombined to produce some other signal. Such structures can cause difficulties for programs that reason about circuit behavior through local propagations. A simple example is shown in Figure 4.8. In the unencapsulated version, purely local propagation cannot deduce from A=1 and C=1 that

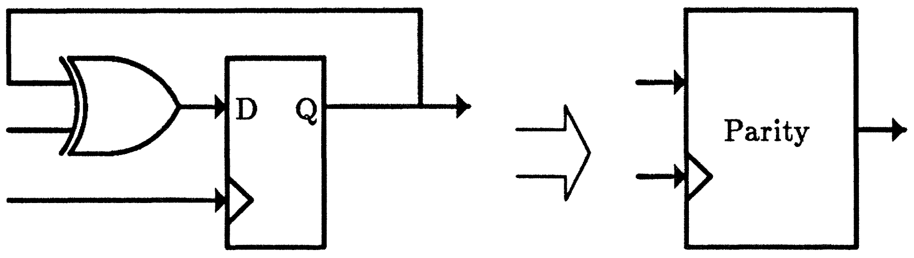
F must be 0. Encapsulating the fanout of B and its reconvergence alleviates the problem.

Figure 4.8: Encapsulating Reconvergence



3. To encapsulate loops. Digital circuits often perform computations sequentially with a loop of combinational circuitry and registers that store intermediate results. The encapsulation of combinational circuitry and registers may have a combined behavior that is simpler to reason about than that for all the individual components. In a sense this is a special case of encapsulating reconvergence. Figure 4.9 shows a simple example; the combined D-flipflop and XOR-gate form a parity generator for a serially encoded input. In concert with the appropriate behavioral abstraction it is not necessary to reason about the clock-by-clock operation of the combined structure.

Figure 4.9: Encapsulating a Sequential Loop



Every non-primitive functional component that appears in the Console Controller Board description is motivated by, and an example of, one of these three principles. The interesting and difficult part of the story concerns the detection and formulation of the appropriate abstract behaviors to go along with the structural compositions; that is treated in the next chapter.

Chapter 5

Representing Circuit Behavior

A central requirement of the model-based troubleshooting methodology is that the program be able to make predictions about behavior based on observations of the inputs and outputs of a device and its subcomponents. Making predictions requires both representation of behavior and computational machinery to determine that, for example, "if A is an adder and its inputs are 2 and 2, its output is 4." In practice, a second requirement is that the program be able to make those predictions using a variety of different domain-specific abstractions, making compromises between the precision and efficiency of predictions made with different vocabularies. In the case of adder A, there might be a good reason to represent the inputs either more abstractly as simply "even" or "odd" or more concretely as bit vectors. Since troubleshooting real digital circuits means reasoning about the behavior of components from resistors to microprocessors, the representation must be flexible enough to integrate many levels of abstraction.

This chapter is partly about TINT, a language of predicates and rules that builds on BASIL by propagating temporal constraints through a network of instantiated components. TINT is a framework that can be used to describe the behavior of components at several levels of detail. What is important, however, is not just the framework itself, but the rich variety of abstractions and component behaviors that will populate it. Hence this chapter is also about the abstractions that make it possible to represent the behavior of complex circuits for troubleshooting.

The primitive level of abstraction is a switch level model that uses voltages in the set {0,1} and currents in the set {-,0,+}. The switch level model is discussed in Appendix E; for the most part the reader may assume that the primitive level of detail is the standard digital model using voltages in the set {0,1}. Some traditional abstractions appropriate to representing and troubleshooting complex digital circuits are those that concern the manipulation of groups of bits — *spatial* abstractions that make it possible to describe (for example) the signal being carried by an 8-bit bus as a number or an ASCII character instead of a bit vector.

Yet, there are much more powerful abstractions; motivating them requires defining some terminology. The purpose of a behavior model in troubleshooting is to make predictions based on observations of the device. The predictions produced by a given model can be characterized as to the *fidelity* with which they match the real world, their *precision*, and the *efficiency* with which they can be made.

Fidelity is best illustrated by a counterexample: suppose a digital mod 16 adder were to be represented as if it did ordinary integer addition. Presenting the real adder with inputs of -8 and -8 correctly produces a 0. The model, however, predicts that it should produce -16. This violates fidelity, and the behavior of the adder in this case would be improperly regarded as a symptom of failure. In model based troubleshooting, fidelity is an overriding goal, since it is better to make an imprecise prediction than to make a wrong one.

Precision and loss of precision in the predictions made with a behavior model are intimately tied to the level of abstraction in the model. For example, modeling the mod 16 adder in terms of the voltages on its input and output wires would be more precise than modeling it using its mod 16 definition. Modeling it in terms of "negative" and "nonnegative" numbers would sacrifice precision in two ways. One of these ways is the loss of precision in the numbers themselves. A second loss of precision occurs because the behavior of the real adder is a total function, but the behavior with respect to "negative" and "nonnegative" is partial, since "negative + nonnegative" yields an ambiguous result. This latter special type of precision loss will be referred to as a loss of *strength*, that is, *weakness* in the behavior model.

The goals of precision and efficiency can be traded off against one another: if efficiency were of no concern, the predictions could always be very precise; conversely, the less precise the predictions are the cheaper it is in general they are to make. The problem of modeling behavior for a given class of devices requires choosing vocabularies and behavior descriptions that retain enough precision that real symptoms will be detectable, yet make efficient prediction possible. The consequence of imprecision is diagnostic indiscriminacy. Thus the issue is, what abstractions will sacrifice the least precision for the most efficiency?

Against this background of fidelity, precision, strength, and efficiency issues, troubleshooting complex digital circuits motivates abstractions that sacrifice temporal precision. Among the salient characteristics of the domain are (i) the gap of several orders of magnitude between the temporal granularity at which events occur in the machine and the temporal granularity at which observations can be made, and (ii) the fact that physical failures in digital circuits are frequently manifest at coarse timescales. These characteristics mean that temporal precision can often be sacrificed without losing the strength needed to detect symptoms. Efficiency is gained through temporal abstractions that make it possible to reason about large numbers of events occurring in the circuit without having to refer explicitly to each one. The vocabulary of temporal abstractions includes familiar concepts such as *change*, *sample*, *duration*, *sequence*, *count*, *cycle*, and *frequency*.

The advantage of temporal abstractions is that when applied to components and groups of components with complex behaviors — even microprocessors — the resulting temporally abstract behaviors can be exceedingly simple. The basic idea is that a given behavior can be usefully abstracted if changes on its inputs always result in changes on its output. Every change of value on the input of an inverter, for example, results in a change of value on its output. Even if that property does not hold, there are still several generic principles for forming useful partial descriptions. For example, a temporally abstract behavior for adders might relate the number of changes on its inputs to the number of changes on its outputs, but there is no interesting relationship for the addition behavior as such: both inputs could change simultaneously in such a way that they cancel each other out. One of the generic principles for forming useful partial descriptions is "holding an input constant," and in this case, if one of the inputs of an adder is held constant all changes on the other input do propagate through. Temporally abstract behavior descriptions will be given for a number of components including gates, counters, and microprocessors.

Although the main purpose of this chapter is to present the details of defining and reasoning with behaviors and temporal abstractions, the underlying "modeling for troubleshooting" theme recurs several times:

- Many of the temporal abstractions to be defined are motivated by the desire to explicitly represent easily-observed features of signals.
- Individual behavior definitions are judged for usefulness on the basis of simplicity and therefore the tractability of the prediction problem in a real troubleshooting session.
- Many of the rules that get included in the model are judged worthwhile because they mention observable signals or can make predictions over long stretches of time.
- TINT itself is deliberately limited in its expressive power, and handles the "frame problem" in a simplistic way — two engineering decisions taken because they keep the troubleshooting engine simple.

With troubleshooting as the ultimate goal, this chapter considers in turn the language TINT, the representation of combinational and sequential behaviors, the explicit representation of temporal abstractions, and techniques for constructing temporally abstract behavior descriptions for complex circuits.

5.1 TINT

The behavior of circuit components is represented using TINT[1], a simple temporal reasoning system in which rules are used to derive facts about the values of functions of time. A function of time is called a *signal*; for example, the voltage at a circuit node is a signal because its value can change over time. In contrast to more sophisticated models of time (for example, the interval model in [Allen84]), for simplicity time is taken to be a sparse set, the integers divisible by a temporal granularity constant δ. Granularity can be thought of as the smallest unit of time that is measurable by available instruments. For the most part the rules and other definitions that follow would remain unchanged for the limit as δ goes to 0 if time were taken to be dense. TINT provides two predicates `thru` and `tsame` for making assertions about signal values:

1. `[thru ?l ?u ?signal ?value]` means that from the lower bound time `?l` to the upper bound time `?u` inclusive, `?signal` had value `?value`.

2. `[tsame ?l ?u ?signal1 ?signal2]` means that at every time between the lower bound `?l` and the upper bound `?u` inclusive, `?signal1` has the same value as `?signal2`.

Any token can appear as the `?value` of a signal.

Only integers, $-\infty$, and $+\infty$ can appear as time arguments to the `thru` and `tsame` predicates. This use of timestamps in TINT rather than symbolic quantities or expressions results in serious limitations as compared to other temporal reasoning systems, but it is adequate for demonstrating troubleshooting.

5.1.1 Signals

The `?signal` arguments of the `thru` and `tsame` predicates are function terms. For example, the term `(voltage (in a u32a))` denotes the voltage signal at port `(in a u32a)`. The `voltage` function maps a port to a real-valued signal. Functions from signals to signals will be used to define *abstractions* and

[1]Timestamped INTervals.

behaviors. Abstractions describe relationships between signals at different levels of detail. Behaviors describe the relationships that components enforce between their input and output signals.

Signals, abstractions, and behaviors are denoted for concreteness as procedures in a side effect free LISP dialect similar to SCHEME [Abelson85], as in [Weise86]. These procedures are not executed by an interpreter; their sole purpose is "mental hygiene": before writing rules to make inferences about the values of signals at various levels of abstraction it is important to know what the signals mean. Almost any other language could have been used, but the essential concepts all concern functions, and SCHEME (and the underlying lambda calculus) is a powerful and familiar representation for functions. Only a few such definitions are shown in this section; the remainder are in Appendix B. All obey the following conventions:

- The `==` symbol indicates definitional equivalence and "..." indicates elision; for example, `x == (lambda (y) ...)` indicates that `x` is a function of one argument whose body is not shown.

- Capitalized symbols denote function arguments and lowercase symbols denote all others.

For example, a function like `voltage` is primitive and can be defined as shown below. It maps a port into a function from time to real numbers:

```
voltage ==
  (lambda (port)
    (lambda (time) ...))
```

The abstraction `voltage-to-logic-level` expects a function of time whose range is the real numbers and returns yet another whose range is {`0`, `1`}:

```
voltage-to-logic-level ==
  (lambda (V)
    (lambda (time)
      (if (< (V time) 1.5) 0 1)))
```

The function `ll` takes a circuit node and yields a function of time whose range is {`0`, `1`}:

```
ll ==
 (lambda (port)
    (voltage-to-logic-level
      (voltage port)))
```

TINT does not use these `lambda` definitions directly, but rather reasons with predicate ground terms containing composite terms built up from primitive signals and abstractions. `[thru -∞ +∞ (ll (in a u32a)) 1]`, for example, means that the logic-level at port `(in a u32a)` was always `1`. `[thru -∞ +∞ (change S) nil]` means that the value of some signal `S` never changed, or, literally, that the signal resulting from the application of the `change` abstraction to signal `S` was always `nil`.

5.1.2 Rules

TINT provides rules that are used in data-driven (forward chaining) fashion to propagate the consequences of observations of signals. The following is a rule as the program would see it. It says that if `x` is a `thing`, and the value of any signal `s` is known over an interval of positive duration, then the signal obtained by applying `abstraction` to `s` is the `fun` of its value:

```
(defmyrule nonsense-rule (:forward)
  :p [isa ≡x thing]
  :s [thru ≡l ≡u ≡s ≡v]
  :f (< ≡l ≡u)
  :l (tell '[thru ,≡l ,≡u [abstraction ≡s] ,(fun ≡v)]))
```

The rules use an extension of JOSHUA syntax. The ≡ prefix marks universally quantified variables; `:p` marks trigger patterns whose matching predicate terms will not appear in any resulting truth maintenance system (TMS) clauses; `:s` marks the predicate terms that do appear in clauses; `:f` marks LISP filters that must return non-`nil` for the rule to fire; `:l` marks the LISP body of the rule; ` starts a quoted structure template and `,` indicates evaluation of a form within that template, as in Common LISP ([Steele84] pp. 349–351). For implementation reasons, there is no distinction between function and predicate terms; they are both denoted with `[ ]` syntax.

For presentation purposes, however, the above rule would be formatted as follows, using ? to indicate variables, omitting details of truth maintenance and backquoting, and retaining for clarity the distinction between predicate and function terms:

```
  If   [isa ?x thing]
 and   [thru ?l ?u ?s ?v]
 and   (< ?l ?u)
Then   [thru ?l ?u (abstraction ?s) (fun ?v)]
```

5.1.3 Signal Histories

The set of all `thru` predications (predicate ground terms) referring to the same signal is called the *history* of the signal. TINT maintains the following invariants for every pair of predications in a given signal history:

- Conciseness: overlapping intervals of the same history are combined into *maximal intervals.* If `[thru ?l1 ?u1 ?s ?v]` and `[thru ?l2 ?u2 ?s ?v]` are both true, and the two intervals touch or overlap — that is, `(+` δ `(max ?l1 ?l2))` is less than or equal to `(min ?u1 ?u2)` — then `[thru (min ?l1 ?l2) (max ?u1 ?u2) ?s ?v]` is also true. The latter predication denotes a maximal interval. As long as it remains true, it *shadows* both predications `[thru ?l1 ?u1 ?s ?v]` and `[thru ?l2 ?u2 ?s ?v]`, and any other predications it subsumes. Rules never fire on shadowed terms.

- Consistency: signals cannot have more than one value at any given time. `[thru ?l1 ?u1 ?s ?v1]` and `[thru ?l2 ?u2 ?s ?v2]` cannot both be true unless either their values are the same (that is, `(equal ?v1 ?v2)`) or the intervals are disjoint (that is, `(< ?u1 ?l2)` or `(< ?u2 ?l1)`). Otherwise TINT records a conflict.

TINT takes advantage of the fact that the lower bound argument `?l` in `thru` predications is restricted to a totally ordered set to organize each signal history as a list ordered by lower bound. This makes the above invariants relatively easy to check and enforce.

A truth maintenance system is used to maintain boolean constraints among `thru` (and other) predications. Ordinary implication is encoded as a clause; for example, if X and Y together imply Z then there is a clause

$\neg X \vee \neg Y \vee Z$. A "shadowed" assertion is one that is implied by other (presumably more general) assertions and that should not trigger rule firings as long as it is implied by those other assertions. Shadowing is implemented as an extension to the TMS. A clause may have any of its literals marked to be shadowed when they are the only satisfiable literal in the clause. For example, suppose A is more general than B. Let $\neg A \vee B$ be a clause with B marked to be shadowed. If A is true, then B is the only satisfiable literal, hence B is true and shadowed. If A were marked to be shadowed as well, then if B is false, A is the only satisfiable literal, so A is false and shadowed. Rules do not fire on predications while they are shadowed.

Figure 5.1 shows a simple example of how the conciseness and consistency invariants are maintained in the history of a signal `S`. Each rectangle indicates a predication and is positioned along the timeline according to the interval that it refers to (each discrete time point is drawn as an interval on the real line). Clauses are indicated by numbered circles; `+` indicates that the attached literal occurs positively, `-` that it occurs negatively, and `()` indicates shadowing. The network is constructed by the following series of operations:

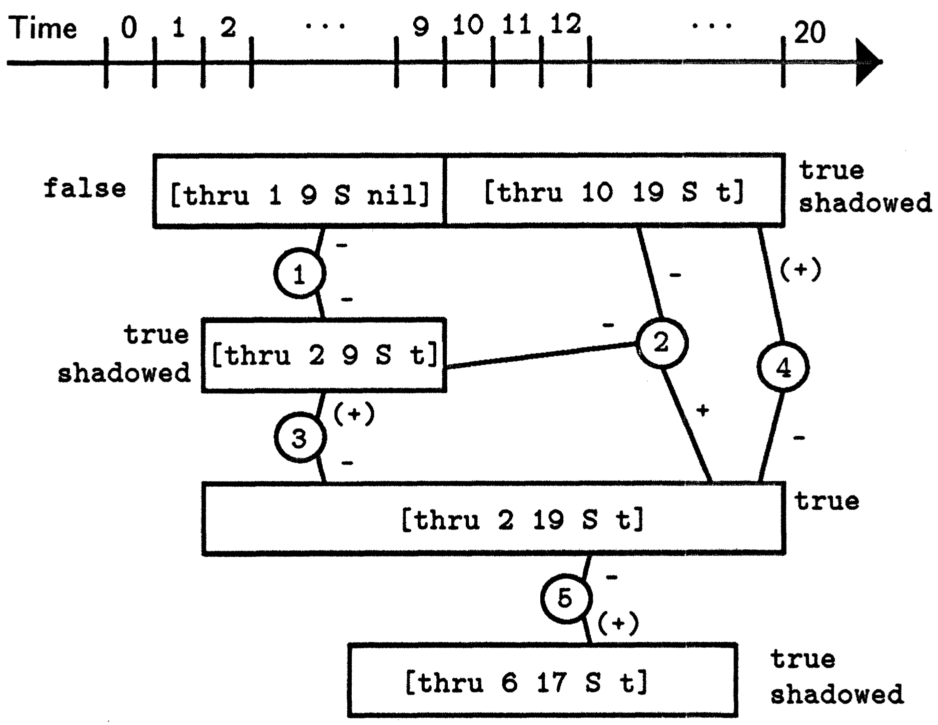
1. Some outside client (the troubleshooting engine, for example) asserts both `[thru 1 9 S nil]` and `[thru 2 9 S t]`. This violates consistency and causes a conflict, represented by clause `1`. The client retracts `[thru 1 9 S nil]`, which the TMS then makes false.

2. The client asserts `[thru 10 19 S t]`, and since it overlaps with `[thru 2 9 S t]` (also true), TINT creates the new predication `[thru 2 19 S t]` and installs clause `2`. Now `[thru 2 19 S t]` subsumes the two predications it depends on, so TINT shadows them by installing clauses `3` and `4`.

3. The client asserts `[thru 6 17 S t]`, but it is immediately shadowed because `[thru 2 19 S t]` subsumes it (clause `5`).

Subsequent retractions and changes of truth value may trigger the creation of new maximal interval predications and new clauses. For example, if the client were now to retract `[thru 10 19 S t]`, the TMS would make `[thru 2 19 S t]` go out, unshadowing `[thru 2 9 S t]` and `[thru 12 17 S t]`. Since the latter two overlap, TINT would then create a new predication `[thru 2 17 S t]` to be created (not shown).

Figure 5.1: TINT Signal History Example



5.1.4 Equality

The behavior of simple components such as wires, buffers, and switches is often easily expressed as a temporally quantified equality; for example, if a switch is closed during the interval from ?l to ?u then during that time the logic-levels at its two terminals will be equal. Also, it is sometimes convenient to give different names to the same signal, so equality between signals is a useful notion as well. The four-place predicate **tsame** captures these concepts; [**tsame** ?l ?u ?s1 ?s2] means that the signals ?s1 and ?s2 had the same value at every time from ?l to ?u inclusive. In the case of different names for the same signal ?l and ?u are $-\infty$ and $+\infty$ respectively.

There are no rules with tsame as a trigger pattern, but TINT does have a demon facility that is used to compute the transitive closure of the congruence relation with respect to tsame assertions and unshadowed thru predications.

For example, if a is equal to b over the interval 5 to 10, then knowing the value of b over any subinterval is propagated to an interval of a:

```
[tsame 5 15 a b]
[thru 0 10 b t]
⇒
[thru 5 10 a t]
```

The consequences of equality of signals a and b are also propagated to their abstractions; hence if a value had been known for (g (f a)) it would get propagated to (g (f b)) as well:

```
[tsame 5 15 a b]
[thru 0 10 (g (f a)) nil]
⇒
[thru 5 10 (g (f b)) nil]
```

This is a brute force technique in at least two respects. When two names refer to the same signal, it would be better to maintain a single canonical name for each signal, or in fact for each function and predicate term, as is done for example in [McAllester80a]. In addition to this redundancy of *facts* the scheme used in TINT also results in redundancy of *derivations*, since the same fact may be derivable in different ways simply by using equalities and other rules in different orders. It would be better to control the invocation of rules so that fewer redundant derivations are created, as is done in BREAD [Feldman88]. The brute force technique used in TINT is only tolerable because the language is restricted to equalities between signals, and the consequences are propagated only for thru predications. If arbitrary terms could be equated, the number of variant terms would quickly explode.

5.1.5 Summary

TINT provides predicates, rules, and a framework of signals and abstractions that together are used to describe circuit behavior. The preceding treatment of TINT is brief because the language itself is not particularly important. The main concern is the vocabulary of signal types and abstractions and the specific rules that the program will use to reason about them. The next three sections will discuss in detail (i) the description and use of combinational (time-independent) behaviors, (ii) the description and use of sequential behaviors, and (iii) abstractions as embodied in TINT along with a particular vocabulary of temporal abstractions.

5.2 Combinational Behaviors

BASIL components have intended behaviors that are functions from signals to signals, and these behaviors can be translated into rules. For example, the intended behavior of a digital inverter is `tinvert`:

```
tinvert == (lambda (S) (lambda (time) (invert (S time))))

invert == (lambda (x) (if (= 1 x) 0 1))
```

This definition can be translated into a rule that asserts facts about the output signal of the inverter based on facts about its input signals.

The intended behavior of a component depends on some collection of background conditions — for example, that the component in question is "working" (not physically damaged), that it is connected to a power source, and so forth. The conditions currently included are those about signals that travel over wires and that are expected to be stable over long periods of time. The condition that there be a 5 volt drop from power to ground is an example, the condition that a clock of a certain constant frequency be provided is another. Conditions relating to other features, such as component temperature, magnetic fields, alpha radiation, and so forth are not included in the model. Failures arising from those sources will be misdiagnosed.

These background conditions must somehow be incorporated into the rules. By convention, the background conditions for a component are collected and summarized as a `mode` signal whose value is `normal` during the intervals that all the conditions are satisfied.

For example, the following rule says that if an adder `?a` is believed to be working, then its mode is normal as long as it is getting power (the `isa` and `status-of` predicates were defined and discussed in Chapter 4):

```
  If  [isa ?a adder]
 and  [status-of ?a working]
 and  [thru ?l ?u (power (in power ?a)) t]
Then  [thru ?l ?u (mode ?a) normal]
```

The principal behavior rule for adders thus depends on the `mode` signal having the value `normal`. In the following rule the signals `(num ...)` denote the signals appearing at the adder ports `(in 0 ?a)`, `(in 1 ?a)`, and `(out 0 ?a)`:

```
    If  [isa ?a adder]
   and  [thru ?l1 ?u1 (mode ?a) normal]
   and  [thru ?l2 ?u2 (num (in 0 ?a)) ?v1]
and[2]  (overlap (?l1 ?u1) (?l2 ?u2))
   and  [thru ?l3 ?u3 (num (in 1 ?a)) ?v2]
   and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
  Then  [thru (max ?l1 ?l2 ?l3) (min ?u1 ?u2 ?u3)
              (num (out 0 ?a)) (+ ?v1 ?v2)]
```

`overlap` tests whether the mentioned intervals have any point in common. The proliferation of "time" variables (six, in this rule) and all the `min`/`max` arithmetic on them may seem like an unfortunate feature of the syntax of TINT. Certainly macros could be written for combinational rules that capture the cliche "the intersection of all the input intervals must be nonempty," as in the rule above. For presentation purposes, this has not been done since there are many sequential behavior rules that defy such simple categorization. It was deemed better to have one general and explicit style of rule presentation than to have multiple incompatible styles.

There are two other rules arising from the behavior definition of the adder, not corresponding to the input/output directionality of the component. These will be called *antibehavior* rules to indicate that their direction of firing is "against" that of causality in the intended behavior of the adder. They look very much like the previous rule, the difference being that the first one below makes deductions about `(in 0 ?a)` and the second about `(in 1 ?a)`:

```
    If  [isa ?a adder]
   and  [thru ?l1 ?u1 (mode ?a) normal]
   and  [thru ?l2 ?u2 (num (out 0 ?a)) ?v1]
   and  (overlap (?l1 ?u1) (?l2 ?u2))
   and  [thru ?l3 ?u3 (num (in 1 ?a)) ?v2]
   and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
  Then  [thru (max ?l1 ?l2 ?l3) (min ?u1 ?u2 ?u3)
              (num (in 0 ?a)) (- ?v1 ?v2)]
```

[2]This condition is semantically redundant, but makes runtime rule matching more efficient.

```
  If  [isa ?a adder]
 and  [thru ?l1 ?u1 (mode ?a) normal]
 and  [thru ?l2 ?u2 (num (out 0 ?a)) ?v1]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (num (in 0 ?a)) ?v2]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
Then  [thru (max ?l1 ?l2 ?l3) (min ?u1 ?u2 ?u3)
            (num (in 0 ?a)) (- ?v1 ?v2)]
```

Figure 5.2 shows how these four rules concerning the adder cooperate to infer signal values, and how they interact with the conciseness condition on TINT signal histories. The network of thru predications shown was created by the following operations:

1. The predications [status-of A working] and [thru 1 80 (power (in power A)) t] are true, so the mode rule of the adder fires and results in the predication [thru 1 80 (mode A) normal], supported by clause 1.

2. The predications [thru 11 50 (num (in 0 A)) 7] and [thru 21 60 (num (in 1 A)) 12] are true, so the behavior rule for the adder fires, resulting in the predication [thru 21 50 (num (out 0 A)) 19] supported by clause 2.

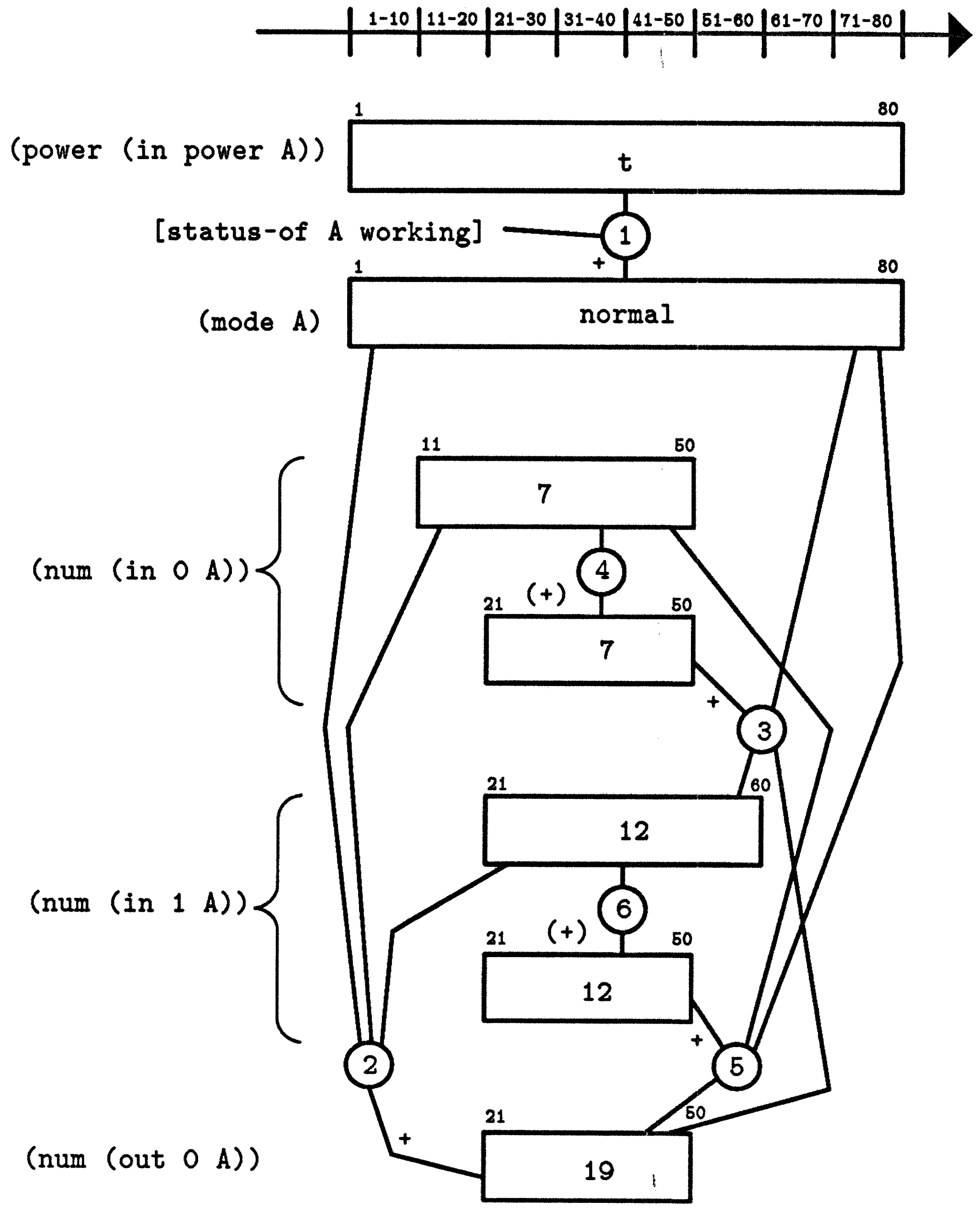
3. The first of the antibehavior rules for the adder fires and deduces [thru 21 50 (num (in 0 A)) 7] by clause 3, but it is shadowed (clause 4) by the enclosing interval.

4. Similarly, the second antibehavior rule fires and deduces [thru 21 50 (num (in 1 A)) 12], which is immediately shadowed.

Were the newly deduced intervals not shadowed, the behavior rule for the adder would fire one more time to deduce [thru 21 50 (num (out 0 A)) 19] again. There is redundancy in this scheme, but without the conciseness condition on signal histories it would be worse.

The behavior rules for the adder serve as a canonical example of the combinational case — the output at any moment is solely a function of its present inputs. The behavior of many other components appearing in

Figure 5.2: Combinational Behavior Example



the Console Controller Board can be expressed in their entirety this way; for other components, portions of their temporally abstract behaviors are combinational in the same sense.

5.3 Sequential Behaviors

The previous examples of behavior rules involved only combinational behaviors. Sequential behaviors require introducing signals to explicitly represent the internal states of components. As with any program for reasoning about change, TINT encounters the *frame problem* [McCarthy69], or, in more illuminating terminology, the *initiation* and *persistence* problems [Shoham86].

The initiation problem arises from the need to specify all the preconditions for a given change or event to occur. The solution in TINT is to explicitly represent whether a component is physically damaged and conditions on incoming electrical signals, summarize them into a `mode` signal, and leave all remaining background assumptions implicit.

The persistence problem arises from the need to specify all the conditions under which nothing happens, that is, the conditions under which states do not change over time. One formal solution is to have minimality criteria (as in [Lifschitz87] and [Shoham86]) that specify which of many possible extensions of an initial set of statements are preferred. An example of such a minimality criterion is to prefer extensions that have the fewest number of changes having no known cause. The validity of any particular prediction is thus relative to many other predictions that have been or could be made. The solution in TINT is to make explicit the persistence conditions for each state. The result is a rule — a frame axiom — for every state signal that mentions every kind of event that could change that state.

Neither of these solutions in TINT are general, since both rely on the belief that each component interacts with few enough other components and in few enough ways that they can all be listed explicitly. Nevertheless, they do have the desirable property that all justifications for signal value predictions are grounded solely in beliefs about the status of components and the observations of the troubleshooter. Having made no appeal to persistence assumptions or any minimality criteria while computing the consequences of observations, each prediction has only local justifications and local consequences. There is thus no need for the detection and manipulation of conflicts to be any different than for combinational behaviors.

A falling-edge triggered register provides the simplest example of sequential behavior, involving only three rules. The first rule says that (a) the output of the register is identical to its state, and that (b) changes from `1` to `0` on the clock input are "interesting:"

```
  If  [isa ?r register]
Then  [tsame -∞ +∞ (state ?r) (num (out 0 ?r))]
 and  [interesting-event (11 (in clk ?r)) (1 0)]
```

The value of the abstract signal `(event ?from ?to ?s)` is `t` whenever there has been a change from the value `?from` to `?to`. The value of this abstract signal is recorded explicitly only when that event type is marked as "interesting." Further details will be presented shortly.

The second rule is a state-transition rule. Any change from `1` to `0` on the clock input causes the register to enter the state selected by its data input signal `(num (input 0 ?r))`. The previous state of the register is irrelevant. The rule below concludes that during (at least) the single moment succeeding the transition, `state` had the value `?input`:

```
  If  [isa ?r register]
 and  [thru ?l1 ?u1 (mode ?r) normal]
 and  [thru ?l2 ?u2 (event 1 0 (11 (in clk ?r))) t]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (num (in 0 ?r)) ?input]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
Then  [thru (+ δ ?u2) (+ δ ?u2) (state ?r) ?input]
```

The third rule is a persistence rule. The register stays in whatever state it is in so long as there has been no change of the clock from `1` to `0`. Its state persists while the event is occurring as well, hence the appearance of δ in the conclusion:

```
  If  [isa ?r register]
 and  [thru ?l1 ?u1 (mode ?r) normal]
 and  [thru ?l2 ?u2 (event 1 0 (11 (in clk ?r))) nil]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (state ?r) ?state]
 and  (<= (max ?l1 ?l2) ?l3 (min ?u1 ?u2))
 and  (not (and (= ?l3 (max ?l1 ?l2))
                (= ?u3 (+ δ (min ?u1 ?u2)))))
Then  [thru (max ?l1 ?l2) (+ δ (min ?u1 ?u2))
            (state ?r) ?state]
```

Figure 5.3 shows these behavior rules in use. The signal denoted `(l1 (in clk R))` is the clock input to a register `R`, and has a history of values `1`, then `0`, then `1`. The predications and clauses were constructed by the following steps:

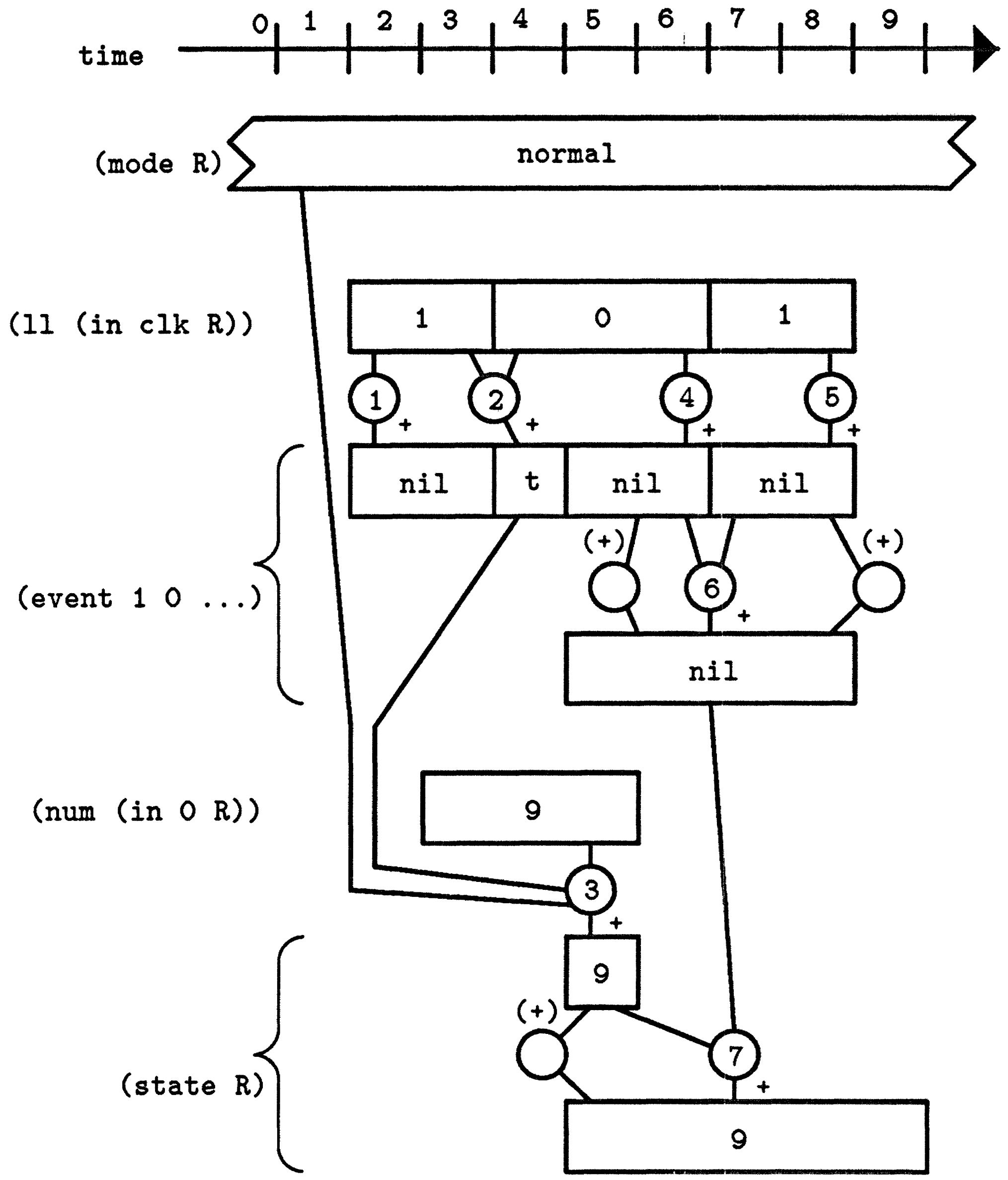
1. Because the change from `1` to `0` has been deemed interesting, the predication `[thru 2 3 (l1 (in clk R)) 1]` results in clause `1` being installed, and similarly for the clauses `2`, `4`, and `5`. Clause `6` is installed to enforce the conciseness condition on the history of `(event 1 0 (l1 (in clk R)))`, and this subsequently results in some predications being shadowed.

2. The transition rule for registers fires and results in clause `3` being installed: the `mode` of the register was normal, the value at `(in 0 R)` was known, and a falling edge occurred on the clock input. The conclusion of the rule is `[thru 5 5 (state R) 9]`.

3. The persistence rule then fires to create clause `7` and the predication `[thru 5 9 (state R) 9]`, which in turn shadows `[thru 5 5 (state R) 9]` to ensure conciseness.

In general, transition rules deduce that a component must have been in a state for just one moment, and the persistence rules subsequently deduce how long that state must have lasted. The rules for the register are particularly simple because at a transition the previous state of the register does not matter; later examples consider cases where it does.

Figure 5.3: Example with Register Behavior Rules



5.4 Abstractions

The notion of an "abstraction" takes on a specific meaning in TINT as a function from signals to signals. Behaviors are functions from signals to signals too, for example, `tinvert` represents the behavior of a boolean inverter. Abstractions and behaviors are not syntactically identical in TINT by accident. Their similarity helps to illuminate the relationship between precision and strength in behavior prediction. Given any abstraction `A` and behavior `B` we can define a function `AB` that describes the abstracted behavior (Figure 5.4). `AB` will usually be a partial function. As long as `A` is not a one-to-one function, the predictions made by `AB` must be less precise than those by `B`. Fidelity requires that any prediction made by `AB` must be the same as that made by `B`; that is, let `z == (B x y)` and then `(A z) == (A (B x y))`:

> For all `times`,
> If `((AB (A x) (A y)) time)` is defined
> Then `((A (B x y)) time) == ((AB (A x) (A y)) time)`

Figure 5.4: Abstractions and Behaviors

$$\begin{tikzcd}
(A\ x)\ (A\ y) \arrow[r, "AB"] & (AB\ (A\ x)\ (A\ y)) \\
 & (A\ z) == (A\ (B\ x\ y)) \\
x\ \ y \arrow[uu, "A"] \arrow[r, "B"] & z \arrow[u, "A"]
\end{tikzcd}$$

The strength of `AB` can be characterized by the degree to which `AB` is a total function. Ideally, any prediction made by `(A (B...))` will also be made by `AB`, as stated below; weakness just means that there are fewer values of `x` and `y` for which it holds:

> For all `times`,
> If `((A (B x y)) time)` is defined
> Then `((A (B x y)) time) == ((AB (A x) (A y)) time)`

For example, let A be the sign function that maps real numbers to {-,0,+}, and let B be real addition. AB is the qualitative addition function qplus, which is partial because (AB + -) and (AB - +) are undefined (Figure 5.5). AB in this case does not yield strong predictions.

Figure 5.5: Example of Abstractions and Behaviors

$$\begin{tikzcd}
\texttt{(sign x)} & \texttt{(sign y)} \arrow[r, "\texttt{qplus}"] & \texttt{(qplus (sign x) (sign y))} \\
 & & \texttt{(sign z) == (sign (plus x y))} \\
\texttt{x} \arrow[uu, "\texttt{sign}"] & \texttt{y} \arrow[uu, "\texttt{sign}"] \arrow[r] & \texttt{z} \arrow[u, "\texttt{sign}"]
\end{tikzcd}$$

Any behavior can be abstracted using any abstraction. Moreover, there is no reason that the same abstraction A need be applied to all the signals x, y, and z. However, for an arbitrary combination of behavior and abstractions, any function AB is unlikely to be strong — that is, its result will be usually undefined — and in fact nearly always useless. An alternative is to make assumptions about the relationship between x and y such that AB is stronger over the resulting restricted domains. In the case of qualitative addition, an example would be to assume that (sign x) and (sign y) are never -, so that the resulting restriction of qualitative addition became a total function.

Every behavior B can also be abstracted trivially to yield strong predictions from the identity behavior I == (lambda (X) X). The "trick" is to have the abstraction of the inputs of B be the procedure B itself (Figure 5.6). All the complexity of the behavior of B has simply been hidden in the abstraction of its inputs. Although this particular abstraction is silly, it is just the extreme example of a more generally useful principle: in trying to formulate a useful behavioral abstraction, some of the behavioral complexity of B can be shifted into the abstractions to make AB simple and strong.

An example is provided by the abstracted behavior of a 4-bit counter that increments on falling edges of its input (Figure 5.7). By temporally abstracting its input and carry-out output with respect to the number of falling

Figure 5.6: Sufficiently Complex Abstractions Make Any Behavior Trivial

$$\begin{tikzcd}
\texttt{(B x y)} \arrow[r, "\texttt{I}"] & \texttt{(I (B x y)) = z} \\
 & \texttt{(I z) = z} \\
\texttt{x y} \arrow[uu, "\texttt{A==B}"] \arrow[r, "\texttt{B}"] & \texttt{z} \arrow[u, "\texttt{I}"']
\end{tikzcd}$$

edges on each signal, the counter can be viewed as dividing the abstracted input by 16. The complicated definition of the abstraction "count of falling edges" textually resembles the definition of the behavior of a counter, so in this case it is in a quite literal sense that some of the behavior `B` has been shifted into the abstraction `A`.

Figure 5.7: The Behavior of a Counter with Respect to a "Counting" Abstraction

$$\begin{tikzcd}
\texttt{(A x)} \arrow[r, "\texttt{divide by 16}"] & \texttt{(/ (A x) 16)} \\
 & \texttt{(A z)} \\
\texttt{x} \arrow[uu, "\texttt{A == count of falling edges}"'] \arrow[r, "\texttt{4-bit counter}"] & \texttt{z} \arrow[u, "\texttt{A == count of falling edges}"']
\end{tikzcd}$$

Good abstractions are not just reformulations of behaviors. Ideally, one has a small collection of abstractions that are appropriate to a wide range of component behaviors — appropriate in the sense of (i) sacrificing precision, (ii) retaining strength, and (iii) increasing efficiency. Given a particular abstraction function `A`, it is thus an interesting and relevant question to ask: for what class of behaviors `B` it is possible to formulate easily computable and strong abstract behaviors `AB`, or, failing that, what reasonable assumptions can be made to strengthen `AB`. Thus characterizing the class of behaviors for which the abstraction is appropriate is a concrete way of characterizing the utility of the abstraction.

5.4.1 Temporal Abstractions

Temporal abstractions are abstractions whose definition mentions previous values of a signal. An example is **`stay`**. **`(stay S)`** is true at **`time`** only if the signal `S` has the same value at `(-` **`time`** δ`)` and **`time`**. The particular temporal abstractions to be shown have the additional property useful in troubleshooting that they produce signals easy to observe in working and malfunctioning circuits. Here, too, **`stay`** is an example: it is often easier to observe whether a given signal is changing than it is to observe the value or values that the signal is taking on.

The breadth of circuits for which these abstractions are appropriate can be briefly characterized as those with behaviors that are event-preserving functions of signals having known relative timing relationships. An *event* is a change in the value of a signal. Behaviors are *event-preserving* to the extent that changes on their input signals are reflected as changes on their outputs (they include all one-to-one functions); three ways that input signals may have a "known timing relationship" are:

1. Behaviors with single inputs, since the timing relationship of a signal with itself is trivial.

2. Behaviors with multiple inputs, all but one of which are constant throughout some interval. Example: the behavior of a two-input and-gate, one of whose inputs is known to be a constant `1` during some interval.

3. Behaviors with multiple inputs for which it can be assumed there are no simultaneous events. Example: the behavior of a two-input xor-gate, whose inputs never rise or fall at the same moment, so that the output always changes whenever the input does. This is a particularly strong assumption to make, and is rarely used.

Having so severely bounded the class of behaviors for which temporal abstractions are useful, it is tempting to conclude that the corresponding class of digital (or other) components is so small as to be worthless. This is not so, because it is possible to structurally compose groups of digital components and define abstract signals in such a way that the behaviors of the resulting aggregate components satisfy those tight requirements. Given that freedom, the relevant class of digital circuit *structures* is so diverse as to defy definition; it is only possible to present examples within that space. After presenting some important temporal abstractions, the next section will be devoted to just such examples. These important temporal abstractions are *change*, *duration*, *sequence*, *count*, *cycle*, *frequency*, and *sampling.*

- *Change* marks events. The **change** function is **t** only at moments when the underlying signal has just changed its value, otherwise it is `nil`. *Stay* is the obvious negation (an example of the values of these signals over time is shown below; it and others like it follow the convention that $\delta = 1$, and that the more abstract the signal the closer it appears to the top line).

(change X)	?	t	nil	nil	t
(stay X)	?	nil	t	t	nil
X	3	4	4	4	5
time	0	1	2	3	4

```
change ==
  (lambda (S)
    (lambda (time)
      (not (equal (S time) (S (- time δ))))))
```

Two familiar numeric elaborations of the **change** abstraction are **dt** (derivative with respect to time) and **cross** (crossings of a value v), defined in Appendix B.

It is also useful to have signals that are **t** whenever a particular event has just occurred and `nil` otherwise. The abstract signal (**event** ?**from** ?**to** ?S) is **t** whenever the underlying signal ?S has just changed from ?**from** to ?**to**. For example, (**event** 500 700 S) is **t** where S has just changed from 500 to 700:

`(event :any 500 S)`	?	?	nil	nil	nil	nil	t
`(event 500 700 S)`	?	?	nil	t	nil	nil	nil
`S`	?	500	500	700	300	700	500
`time`	0	1	2	3	4	5	6

A `?from` argument of `:any` denotes the special case of any transition to `?to`, which is useful for marking the known beginning of an interval. `(event :any 500 S)` is `t` at time `6`. However, it is not known to be `t` at time `1` since the value of `S` could have been `500` at `0`.

In the domain of troubleshooting circuit boards, it is much easier to observe whether a given single-bit signal changed or not during an interval of several seconds than it is to observe each individual change. The abstraction *changing with-respect-to* is specifically tailored to making statements about whether a given logic level signal ever changed, statements that typically arise from observations of the circuit. `(changing-wrt ?l ?u ?S)` is `t` only at the upper bound time `?u` and only when `?S` changed at least once during the interval from `?l` to `?u` inclusive:

`(changing-wrt 1 6 S)`	nil	nil	nil	nil	nil	nil	t	nil
`(changing-wrt 1 3 S)`	nil	nil	nil	nil	nil	nil	nil	nil
`(change S)`	?	nil	nil	nil	t	nil	nil	nil
`S`	0	0	0	0	1	1	1	1
`time`	0	1	2	3	4	5	6	7

For example, if `[thru 6 6 (changing-wrt 1 6 S) t]` is true it means that `S` changed at least once between times 0 and 6.

- *Duration* indicates how long a signal has stayed at the same value. The duration is defined to be δ when the signal has just changed.

`(duration X)`	?	1	2	1
`X`	3	4	4	5
`time`	1	2	3	4

- *Count* counts the number of events that have occurred with respect to a window of fixed width. The function `count-ww` takes an argument `n` that is the width of the window in units of δ, and a signal argument `S`.

`(count-ww 3 S)`	?	?	1	1	2	2	1	1	1
`S`	t	nil	nil	t	t	nil	nil	t	nil
`time`	0	1	2	3	4	5	6	7	8

- The *Sequence* abstraction indicates when a particular string of (possibly repeated) values has appeared contiguously on a signal. Given a sequence like `(0 1)` it can be thought of as a finite string recognizer for occurrences of the regular expression $0^+ \; 1^+$.

`(sequence '(0 1) S)`	nil	nil	nil	t	nil	nil	t	nil	t
`S`	0	1	1	0	0	1	0	1	0
`time`	0	1	2	3	4	5	6	7	8

- The *Cycle* abstraction is used to count the number of endings of a particular sequence of values. The function `cycles-ww` is simply the composition of the *count* and *sequence* abstractions:

`(cycles-ww 3 '(0 1) S)`	?	?	?	1	2	1	2	1	2
`S`	0	1	0	1	0	1	0	1	0
`time`	0	1	2	3	4	5	6	7	8

Typically, the larger the window, the less relative fluctuation of the cycle count over time. For example, suppose `A` and `B` are signals that are just slightly out of phase. `(cycles-ww n ... A)` and `(cycles-ww n ... B)` will have the same value most of the time, and will never differ by more than `1`.

`(cycles-ww 8 .. A)`	2	2	3	2	2	3	2	2	3
`(cycles-ww 8 .. B)`	2	3	2	2	3	2	2	3	2
`(sequence .. A)`	nil	nil	t	nil	nil	t	nil	nil	t
`(sequence .. B)`	nil	t	nil	nil	t	nil	nil	t	nil
`time`	0	1	2	3	4	5	6	7	8

The larger the window, the less the relative difference, and conversely, the easier to detect significant deviations (as for example the difference between a signal occasionally asserted and one that is running at about 20 Khz). By convention, the window size is usually taken to be 1000 times the expected period of the signal, so that the `cycles-ww` of a pair of signals can be judged as equal if they differ by no more than $\frac{1}{10}\%$, that is, by no more than one cycle in a thousand.

- *Frequency* is simply the number of cycles that occurred during a window, divided by the duration of that window. The abstraction function `fww` yields the frequency of a signal with respect to a window size and a particular sequence of values. With a sufficiently large window relative to the cycle time (e.g. 1000 times as large), the result is an adequate approximation to the normal notion of "frequency."

`(fww 3 '(0 1) S)`	?	?	?	1/3	2/3	1/3	2/3	1/3	2/3
`S`	0	1	0	1	0	1	0	1	0
`time`	0	1	2	3	4	5	6	7	8

Sometimes it is not necessary to know the actual frequency of a signal, but simply whether the signal is changing or not. This can be represented as the sign of the frequency.

- The notion of *Sampling* is essential to understanding behavior of synchronous systems; here, the sampling of a signal refers to the values that the signal takes on at certain (usually regularly spaced) moments. The abstraction function `sample-and-hold` (abbreviated `samp`) takes two argument signals `V` and `S`; `V` is `t` where the signal `S` is to sampled. The value of `samp` is the value of `S` where `V` was last `t`:

`(samp V S)`	?	1	1	1	1	0	0	0
`V`	nil	t	nil	nil	nil	t	nil	nil
`S`	1	1	0	1	0	0	0	1
`time`	0	1	2	3	4	5	6	7

Note that the value of `(samp X X)` – the sampling of a signal with itself – at `time` is the value of `X` the last time `X` was non-`nil`.

The interesting and important property of these temporal abstractions is that they sacrifice precision without sacrificing the ability to detect faulty behavior. In troubleshooting the idea is to detect discrepancies between the observed behavior of the real device and our idealized model of it; thus the predictions of interest are those that can be made efficiently from what we have observed and that could be significantly violated if the device were broken. The *change* abstraction is useful because it is easy to observe whether signals in a device are changing or not, and easy to predict what the consequences of change (or lack of it) would be. Similarly, the *frequency* abstraction is useful even if frequencies are hard to observe accurately: the distinction between zero and nonzero frequencies is easy to observe and is likely to result in significantly different behavioral consequences. By summarizing (possibly very long) sequences of events, temporal abstractions make complex behaviors look simple enough for troubleshooting to be tractable.

Abstractions define how a signal such as `(ll n48)` (the logic level at node 48) relates to signals "below" it such as `(voltage n48)`, and signals "above" it such as `(fww` 10^6 `'(0 1) (ll n48))` (the frequency at node 48, measured at cycles starting with 0 and with a window of 10^6 δ time units). Abstractions thus result in rules that can fire "upward," "downward," or even "sideways" between different abstractions of the same base signal. The definitions of `change` and `stay`, for example, can yield the following rules:

```
  If   [thru ?l ?u (stay ?s) ?v]
Then   [thru ?l ?u (change ?s) (not ?v)]

  If   [thru ?l ?u (change ?s) ?v]
Then   [thru ?l ?u (stay ?s) (not ?v)]
```

In practice, however, only a subset of the possible rules should actually be made explicit and included in the program. For example, only one of the signals `(change S)` and `(stay S)` really needs to be represented explicitly, so these two rules are not necessary.

Furthermore, each rule should not be fired on every signal — for example, the *change* abstraction applies in principle to every signal, but if every change of value on the signal `S` required an explicit deduction about `(change S)`, an infinite regress would result — `(change (change S))`, and so forth. The `changing-wrt` abstraction is an example of the general phenomenon that for any given signal there are infinitely many abstractions that can applied to it. For example, (changing-wrt 1 5 S), (changing-wrt 2 7 S), and so on, are all legitimate signals. Some criterion is needed for determining which of the many possible signals TINT will make deductions about. Consequences concerning **changing-wrt** and similar abstractions are only propagated during the interval during which observations are currently being made. TINT denotes the interval over which observations are currently being made in terms of the lower and upper bounds of a global reference interval. By convention the pseudo-signal GR denotes this global reference timeline; it is t during the interval that the circuit is actually observed. Thus, [thru ?a ?z GR t] means that observations are made with respect to the time interval ?a to ?z inclusive. The interval ?a to ?z is referred to as the "observation interval." The pattern [thru ?a ?z GR t] appears in a rule to ensure that it makes its deductions only during the current observation interval.

Furthermore, only certain signals in a circuit are actually observable. Again by convention, the only observable signals are taken to be those at the solder joints of a circuit board, which are modeled in BASIL as the ports of wire etches. Ports of etches are called holes (chip pins are placed into them), and (hole ?i ?e) denotes the ?ith in etch ?e. For efficiency, the rules dealing with observations of signals are restricted to making inferences at these ports.

The result of these conventions and efficiency considerations is that the following four rules suffice to make inferences among a logic-level signal and its **change** and **fww** abstractions:

If the logic-level at a hole has a constant value over the interval from ?a to ?z, then the signal was never changing with respect to a subinterval of observation:

```
  If  [thru ?a ?z GR t]
 and  [thru ?l ?u (ll (hole ?n ?e)) ?v]
 and  (<= ?l ?a ?z ?u)
Then  [thru ?z ?z (changing-wrt ?a ?z (ll (hole ?n ?e))) nil]
```

If the logic-level at a hole had two different values at different moments during an observation interval ?a to ?z, then the signal is changing with respect to that interval:

```
  If   [thru ?a ?z GR t]
 and   [thru ?l0 ?u0 (ll (hole ?n ?e)) 0]
 and   (overlap (?a ?z) (?l0 ?u0))
 and   [thru ?l1 ?u1 (ll (hole ?n ?e)) 1]
 and   (overlap (?a ?z) (?l1 ?u1))
Then   [thru ?z ?z (changing-wrt ?a ?z (ll (hole ?n ?e))) t]
```

The frequency of a signal with respect to a window `?w` implies whether or not it should be changing with respect to the observation interval `?a` to `?z` (provided that the window `?w` fits within the observation interval); if the frequency is nonzero then the signal should be changing, otherwise not:

```
  If   [thru ?l ?u (fww ?w ?seq ?s) ?f]
 and   [thru ?a ?z GR t]
 and   (<= ?l ?a ?z ?u)
 and   (<= ?w (- ?z ?a))
Then   [thru ?z ?z (changing-wrt ?a ?z ?s) (< 0 ?f)]
```

A signal that is not changing has a frequency of 0 with respect to any window and sequence. The following rule says that if the logic-level signal at a hole is not changing during an observation interval, its frequency is 0 during that interval. An additional condition is that there must have been some previous mention of the frequency of that logic-level signal, otherwise irrelevant frequencies would be deduced for many other signals:

```
  If   [thru ?z ?z (changing-wrt ?a ?z (ll (hole ?n ?e)))]
 and   [thru ?a ?z GR t]
and[3] Signal (fww ?w ?seq (ll (hole ?n ?e))) exists
Then   [thru ?a ?z (fww ?w ?seq (ll (hole ?n ?e))) 0]
```

Finally, a noteworthy relationship that will appear implicitly in other rules is that a signal `?s` sampled with respect to some signal `?v` cannot be changing unless the underlying signals are:

```
  If   [thru ?z ?z (changing-wrt ?a ?z (samp (fall ?v) ?s)) t]
Then   [thru ?z ?z (changing-wrt ?a ?z ?s) t]
 and   [thru ?z ?z (changing-wrt ?a ?z ?v) t]
```

[3]This trigger pattern is implemented with a predicate not mentioned elsewhere: `[cohistorical (fww ?w ?seq (ll (hole ?n ?e)))]`

Every signal has many possible `(event ...)` abstractions, but those that will help other behavior rules to fire are worth making explicit deductions about. The predicate `interesting-event` indicates which signals these are. The predication `[interesting-event ?s (?from ?to)]` means that if a change from `?from` to `?to` occurs on signal `?s`, then `(event ?from ?to ?s)` should be `t`:

```
  If   [interesting-event ?signal (?from ?to)]
 and   [thru ?l1 ?u1 ?signal ?from]
 and   [thru ?l2 ?u2 ?signal ?to]
 and   (<= ?u1 ?l2 (+ δ ?u1))
Then   [thru ?l2 ?l2 (event ?from ?to ?signal) t]
```

When `from` is the token `:any`, `(event :any to s)` is `t` no matter what the previous value of `s` was:

```
  If   [interesting-event ?signal (:any ?to)]
 and   [thru ?l1 ?u1 ?from ?signal]
 and   [thru ?l2 ?u2 ?to ?signal]
 and   (not (equal ?from ?to))
 and   (<= ?u1 ?l2 (+ δ ?u1))
Then   [thru ?l2 ?l2 t (event ?from ?to ?signal)]
```

Otherwise, `(event ?from ?to ?s)` should be `nil`. For any interval during which `?s` was constant, either (i) `s` had the value `?from`, in which case there could not have been any such event:

```
  If   [interesting-event ?signal (?from ?to)]
 and   (not (eql ?from :any))
 and   [thru ?l ?u ?s ?v]
 and   (equal ?v ?from)
Then   [thru ?l ?u (event ?from ?to ?s) nil]
```

Or[4], (ii) `?s` had some value other than `?from`, in which case no such event could have happened during the interval starting δ after the beginning and ending δ after the end:

[4]The current implementation treats these two cases with a single rule.

```
  If  [interesting-event ?signal (?from ?to)]
 and  (not (eql ?from :any))
 and  [thru ?l ?u ?s ?v]
 and  (not (equal ?v ?from))
Then  [thru (+ δ ?l) (+ δ ?u) (event ?from ?to ?s) nil]
```

(Figure 5.3 on Page 106 showed the above rules about events in use.)

5.4.2 Composite Abstractions

Composite abstractions involve spatial as well as temporal abstraction. For example, an eight-bit parallel signal is a composite of eight one-bit logic-level signals. BASIL provides the predicate `[corr ...]` that indicates where a port corresponds to an abstraction of one or more subports. Rules that concern composite signals all trigger on occurrences of such correspondences. For example, `[corr ttl-power Z X Y]` means that there is a correspondence of type `ttl-power` between the composite port `Z` and the two ports `X` and `Y`. The type of the correspondence between the ports implies one or more abstraction relationships between signals at those ports. The `ttl-power` correspondence, for example, implies that the abstract signal `(power Z)` is equivalent to the signal `(one-and-zero (ll X) (ll Y))`, where:

```
one-and-zero ==
  (lambda (A B)
    (lambda (time)
      (and (eql (A time) 1) (eql (B time) 0))))
```

A power input of `t` is just shorthand for having the appropriate voltage drop between the power and ground inputs to the device. The following rule says that if a component has power then its power and ground are logic-levels `1` and `0` respectively:

```
  If  [corr ttl-power (in power ?a) ?p ?g]
 and  [thru ?l ?u (power (in power ?a)) t]
Then  [thru ?l ?u (ll ?p) 1]
 and  [thru ?l ?u (ll ?g) 0]
```

In principle, rules could be written to enforce many relationships between the composite signal and its subsignals; only a few concerning the temporal abstraction *frequency* have yet been implemented.

The abstraction `two-phase-clock`, for example, yields rules relating the frequency occurring on the two-phase clock to the frequencies of its subsignals. If the frequency of a two-phase clock signal is nonzero then each of the underlying signals have that same frequency. Since the underlying signals are out of phase, one of them has its frequency measured with respect to the cycle `'(0 1)` and the other with respect to `'(1 0)`; which is which depends on whether the two-phase clock frequency was measured with respect to `'(nil t)` or `'(t nil)`:

```
  If  [corr two-phase-clock-encoding ?clk ?c1 ?c2]
 and  [thru ?l ?u (fww ?w '(?b ?a) (cc ?clk)) ?f]
 and  (< 0 ?f)
Then  [thru ?l ?u (fww ?w (if ?b '(1 0) '(0 1)) (ll ?c1)) ?f]
 and  [thru ?l ?u (fww ?w (if ?b '(0 1) '(1 0)) (ll ?c2)) ?f]
```

Conversely, if the frequency of either subsignal is zero then the frequency of the composite signal is zero as well:

```
  If  [corr two-phase-clock-encoding ?clk ?c1 ?c2]
 and  [thru ?l ?u (fww ?w '(?a ?b) (ll ?c2)) 0]
Then  [thru ?l ?u (fww ?w (if (eql 0 ?a) '(nil t) '(t nil))
                          (cc ?clk)) 0]

  If  [corr two-phase-clock-encoding ?clk ?c1 ?c2]
 and  [thru ?l ?u (fww ?w '(?a ?b) (ll ?c1)) 0]
Then  [thru ?l ?u (fww ?w (if (eql 1 ?a) '(nil t) '(t nil))
                          (cc ?clk)) 0]
```

A similar relationship holds between a synchronous serial signal and the pair of one-bit logic-level signals that comprise it, denoted by the correspondence `clocked-serial`. In this case, the frequency of the serial signal — as measured by the rate of zero-crossings — can be used to determine whether the underlying logic-level signals are changing. The essential relationship is that the frequency of zero crossings on the composite signal must be less than the frequency of the underlying serial data signal sampled with respect to the clock:

- If [corr clocked-serial ?s ?d ?c], then:

```
                                  (fww w '(nil t)
                                    (change
(fww w '(nil t)                 <     (samp
  (cross 0 (cs ?s)))                    (fall (ll ?c))
                                        (ll ?d))))
```

From the fact that both (ll ?c) and (ll ?d) must be changing for (samp (fall (ll ?c)) (ll ?d)) to be changing, this relationship can be used to form the following rule, which says that if the frequency of the composite signal is positive during the observation interval, then both the clock and data signals are changing:

```
  If  [corr clocked-serial ?s ?d ?c]
 and  [thru ?l ?u (fww ?w '(nil t) (cross 0 (cs ?s))) ?f]
 and  (< 0 ?f)
 and  [thru ?a ?z GR t]
 and  (<= ?l ?a ?z ?u)
Then  [thru ?z ?z (changing-wrt ?a ?z (ll ?c)) t]
 and  [thru ?z ?z (changing-wrt ?a ?z (ll ?d)) t]
```

Conversely, if either of the underlying signals are not changing then the frequency of the abstract signal must be zero:

```
  If  [corr clocked-serial ?s ?d ?c]
 and  [thru ?l ?u (ll ?d) ?v]
Then  [thru ?l ?u (fww ?w '(nil t) (cross 0 (cs ?s))) 0]

  If  [corr clocked-serial ?s ?d ?c]
 and  [thru ?l ?u (ll ?c) ?v]
Then  [thru ?l ?u (fww ?w '(nil t) (cross 0 (cs ?s))) 0]
```

A more complex version of the relationship between (cs ?s) and its subsignals applies to multi-bit parallel buses. If the signal on an n-bit bus is known to be changing in such a way that the different values it takes on include both values below and above 2^{n-1}, then the most significant bit must be changing:

```
    If  [corr bus-with-csl-and-wrl ?bus
               ?cs ?wr ?msb .  ?others]
   and  [thru ?l ?u (fww ?w '(nil t) (cross ?n (cp ?bus))) ?f]
   and  (< 0 ?f)
   and  (eql ?n (expt 2 (length ?others)))
  Then  [thru ?z ?z (changing-wrt ?a ?z (ll ?msb)) t]
```

The 12-bit bus in the Audio Decoder, for example, carries 12-bit values, and if the frequency of crossings of 2^{11} on that bus is nonzero, then the most significant bit of the bus must be changing.

5.4.3 Summary of Abstractions

In TINT, abstractions are functions from signals to signals, and in principle any abstraction can be applied to any signal or signals to produce yet another signal. TINT can represent the time varying values of any of these signals, and uses rules to make inferences about signals at other levels of abstraction. Temporal abstractions are among the most useful because temporally abstract signals are among the easiest for the troubleshooter to observe. Using this as a guiding principle, the rules that map between levels of abstraction are written for the most part so as to limit the inferences about signals to those that are observable.

The real utility of the temporally abstract signals, however, is that it is possible to reason about the behavior of circuit components using them. But faced with a digital circuit and the above collection of temporal abstractions, it is not always obvious how the behavior of the circuit should be described with those abstractions, nor even which portions lend themselves to such a description. This model-building process is not automated, but can be metaphorically understood as "parsing" the circuit schematic: grouping components into composite structures and abstracting signals, sometimes hiding them completely. An essential ingredient of the parsing is "knowing what the circuit is for," that is, its purpose. Heavy use of teleological knowledge is made throughout the entire parsing description. The other essential ingredient is "knowing what the model is for." The model is for troubleshooting, and heavy use of that fact is made too. The four basic principles by which behaviors are temporally abstracted are:

1. Event Preservation — some component behaviors lend themselves to temporal abstractions without modifications or new assumptions.
2. Reduction — a temporally abstract behavior that only covers part of a component behavior is better than not covering any at all.
3. Synchronization — some digital circuits have signals that provide timing information, and the *sampling* abstraction can simplify the behavior of components to which they are connected.
4. Encapsulation — after grouping components together, their combined behavior may lend itself to temporal abstraction using the previous two techniques even if the individual component behaviors did not.

These principles are treated individually in the following four sections.

5.5 Event Preservation

A behavior is *event preserving* to the extent that certain types of changes on its input signals result in changes on its output signal. All one-to-one functions are perfectly event preserving and result in abstracted behaviors that are strong. The **tinvert** function that describes the behavior of a boolean inverter is a simple example (Page 98). The **tinvert** behavior is event preserving because it is a one-to-one function. Abstracting the behavior of a one-to-one function with the abstraction **stay** always results in the identity function. In particular, **(stay (tinvert X)) == (identity (stay X))**.

The abstracted behavior **identity** with respect to **stay** is not by itself a useful result, but similar derivations apply to the **cycles-ww** and **fww** abstractions:

```
(cycles-ww n '(0 1) S)  ==  (cycles-ww n '(1 0) (tinvert S))
(cycles-ww n '(1 0) S)  ==  (cycles-ww n '(0 1) (tinvert S))

      (fww n '(0 1) S)  ==  (fww n '(1 0) (tinvert S))
      (fww n '(1 0) S)  ==  (fww n '(0 1) (tinvert S))
```

The identity behavior that results from the **fww** abstraction is useful because predictions about the frequencies of signals can be made over long intervals of time, summarizing many underlying events without having to refer to each one individually.

These relationships between the logic-levels at the inputs and outputs of inverters are simple to encode in rules. The inverter has a rule (like all boolean gates) that says if it is working and it has power, then its mode is **normal**:

```
  If  [isa ?x inverter]
 and  [status-of ?x working]
 and  [thru ?l ?u (power (in power ?x)) t]
Then  [thru ?l ?u (mode ?x) normal]
```

The behavior and antibehavior of the inverter can be captured in two rules:

```
  If  [isa ?x inverter]
 and  [thru ?l1 ?u1 (mode ?x) normal]
 and  [thru ?l2 ?u2 (ll (in a ?x)) ?v]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
             (ll (out y ?x)) (- 1 ?v)]

  If  [isa ?x inverter]
 and  [thru ?l1 ?u1 (mode ?x) normal]
 and  [thru ?l2 ?u2 (ll (out y ?x)) ?v]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
             (ll (in a ?x)) (- 1 ?v)]
```

The behavior of the inverter, being a one-to-one function, is event preserving, and there are potentially several temporal abstractions appropriate for describing its behavior. Rules could be written for the inverter using the abstractions **change**, **stay**, **cycles**, **fww** and so forth, but **changing-wrt** is chosen because it refers to easily observable abstract signals. The rule about whether the signal is changing simply says that during the observation interval, the output is changing if and only if the input is changing:

```
  If  [isa ?x inverter]
 and  [thru ?l ?u (mode ?x) normal]
 and  [thru ?a ?z GR t]
 and  (<= ?l ?a ?z ?u)
Then  [tsame ?l ?u (changing-wrt ?a ?z (ll (in a ?x)))
                   (changing-wrt ?a ?z (ll (out y ?x)))]
```

An inverter can be used to implement a "frequency buffer." The input and output frequencies of a frequency buffer are the same. However, when the underlying signal has been inverted, incoming `'(0 1)` cycles come out as `'(1 0)` cycles, and the rule must take this into account. The following rule says that the frequency of the output with respect to a particular cycle is the same as the frequency with respect to the inverse of that cycle; the rule does not fire unless there has been some mention of the relevant input signal frequency and cycle:

```
  If  [isa ?d frequency-buffer]
 and  [thru ?l1 ?u1 (mode ?d) normal]
 and  Signal (fww ?w '(?a ?b) (ll (in a ?d))) exists
Then  [tsame ?l1 ?u1 (fww ?w '(?a ?b) (ll (in a ?d)))
                     (fww ?w '(?b ?a) (ll (out y ?d)))]
```

A pair of inverters may also form a frequency buffer for a two-phase clock signal. However, the effect of the inversion of the underlying signals is to make the output cycle start a quarter phase later than the input cycle. Over a large number of cycles the phase shift makes little difference in the frequency. Thus the rule says that the frequencies of the cc signals at the input and outputs are the same, provided that there has been some mention of the input frequency:

```
  If  [isa ?d frequency-buffer]
 and  [thru ?l1 ?u1 (mode ?d) normal]
 and  Signal (fww ?w '(?a ?b) (cc (in a ?d))) exists
Then  [tsame ?l1 ?u1 (fww ?w '(?a ?b) (cc (in a ?d)))
                     (fww ?w '(?a ?b) (cc (out y ?d)))]
```

A larger and more interesting class of behaviors than one-to-one functions are those for which a subset of input events always result in some output event. For example, a toggle is a flip-flop that changes its state on every falling edge of its clock input. This behavior can be described with the function **toggle**, which is event preserving with respect to falling edges. Its input, ranging over {0, 1}, has two possible events — rising and falling edges. In any sequence of input events, a fixed subset (about half) will be falling edges. Whenever a falling edge occurs on the input, the output has either a rising or falling edge.

(toggle L)	0	1	1	1	1	0	0
L	1	0	0	1	1	0	0
time	0	1	2	3	4	5	6

It would be useful to have a strong temporally abstract version of the **toggle** behavior; the problem is finding a temporal abstraction that will work. **stay** does not work, but **cycles-ww** does. As noted earlier, any behavior can be combined with any abstraction to yield an abstracted behavior. Unlike one-to-one functions, partially event preserving behaviors abstracted with `stay` do not yield strong functions. For example, by a derivation similar to that for `tinvert`, all that can be shown is that for all `times`, `((stay S) time)` $\rightarrow$ `((stay (toggle S)) time)`, that is, the output never changes if the input does not. This is not strong, because it makes no prediction if the input *is* changing. Partially event preserving behaviors may, however, yield strong functions when abstracted with temporal abstractions other than `stay`. In the case of `toggle` in particular, the behavior derived for the `cycles-ww` abstraction is a total function, by using the additional fact that the value of `S` is `1` or `0` at all `times`: the count of occurrences of the sequence `l='(0 1)` or `l='(1 0)` on the output is approximately half that on the input[5]:

```
        (* 2 ((cycles-ww n l (toggle S)) time))  ≤
                        ((cycles-ww n l S) time)  ≤
    (+ 1 (* 2 ((cycles-ww n l (toggle S)) time)))
```

By substitution using the definition of `fww`, for a sufficiently large value of `n` the following approximate relation holds at all `times`:

```
(* 2 ((fww n l (toggle S)) time))  ==  ((fww n l S) time)
```

Event preservation is not a property solely of a behavior; if the behavior is not a one-to-one function it might be necessary to make use of additional information about the input signal to the function. This may be either through an assumption about the signal, or (as in the frequency divider case) through an intrinsic property the signal possesses by virtue of its type.

`toggle` behaves as a divider with respect to the signal abstraction `fww`; components with the `toggle` behavior can thus be viewed as frequency dividers. Similarly, cascades of components having the toggle behavior — counters, that is — can be viewed as frequency dividers as well, for divisions by powers of 2.

[5]Briefly, the derivation considers four cases on `S`: `1` followed by `1` at `time`, `1` followed by `0` at `time`, and so forth. By using the definition of `toggle` in each case it can be shown that `(cycles-ww n l (toggle S))` must increment by at least 1 for every 2 increments of the signal `(cycles-ww n l S)`.

This is a useful way of viewing the behavior of toggles and counters because sometimes their inputs have known frequencies that are stable over long intervals of time. One way that the frequency of the input signal could be known over a long interval is if it the output of an oscillator. For example, the crystal oscillator in the Console Controller Board generates a 9.8Mhz signal. This is approximated as a frequency of 10^7 cycles per second, with a window size of a thousand periods, that is, $1000 \times \frac{1}{10^7}$ seconds:

```
  If  [isa ?o oscillator]
 and  [thru ?l ?u (mode ?o) normal]
Then  [thru ?l ?u (fww 10^-4sec '(0 1) (ll (out 0 ?o))) 10^6]
```

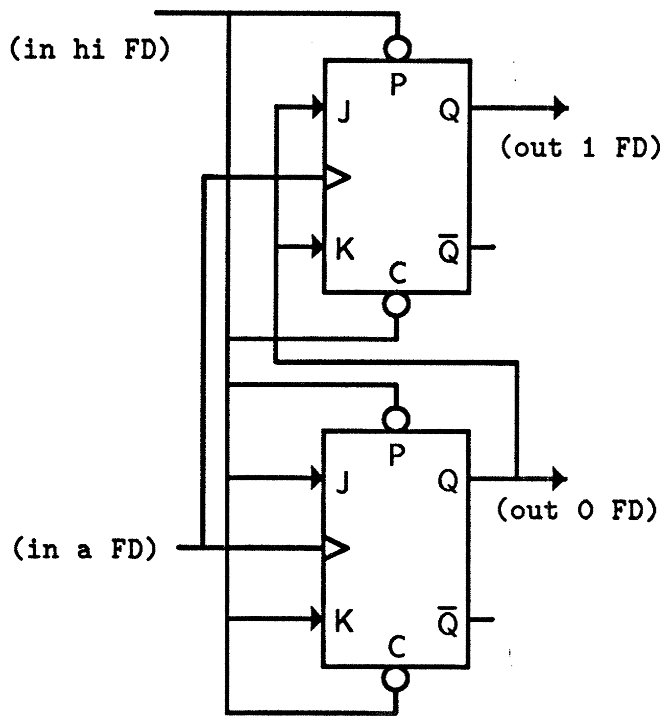
The behavior of the frequency divider allows the program to predict what the output frequency will be over those same long intervals of time. Expressing its behavior in rules introduces some subtleties.

The first subtlety is that until now "power" has been the only input that components required to be in `normal` mode. The frequency divider requires a separate constant `1` input. For example, the Console Controller Board contains several frequency dividers, implemented with one or more JK flipflops or with counters, and one thing they all have in common is that they have some of their inputs pulled up to a constant logic-level of `1`. Figure 5.8 shows an example; the input `(in hi FD)` is tied to several JK flipflop inputs. With both J and K tied to `1`, the flipflop toggles its state on each falling clock edge, and with the Preset and Clear inputs tied to `1` this is the only way it can change its state. The rule for the `mode` of the frequency divider thus includes the condition that the input `(ll (in hi ?d))` must be `1`:

```
  If  [isa ?d frequency-divider]
 and  [status-of ?d working]
 and  [thru ?l1 ?u1 (power (in power ?d)) t]
 and  [thru ?l2 ?u2 (ll (in hi ?d)) 1]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2) (mode ?d) normal]
```

The second subtlety is that frequency dividers can be composed of a cascade of toggle behaviors (a ripple counter can be viewed this way) and hence have multiple outputs, which by convention are numbered from 0 upwards. The frequency at the nth output is thus $\frac{1}{2^{n+1}}$ that of the input.

Figure 5.8: Frequency Divider Implemented with JKFFs



The third and final subtlety is that signals at lower frequencies have longer periods and hence require a longer duration to go through 1000 cycles; the effect is that the window size at the nth output of a frequency divider scales by 2^{n+1}. As a result, a single behavior rule for frequency dividers works for any number of output ports, and makes deductions at different window sizes on those different ports:

```
  If  [isa ?d frequency-divider]
 and  [has-port ?d (out ?n ?d)]
 and  [thru ?l0 ?u0 normal (mode ?d)]
 and  [thru ?l1 ?u1 ?f (fww ?w ?cyc (ll (in a ?d)))]
 and  (overlap (?l0 ?u0) (?l1 ?u1))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
            (fww (truncate (* ?w (expt 2 (+ 1 ?n))))
                 ?cyc (ll (out ?n ?d)))
            (/ ?f (expt 2 (+ 1 ?n)))]
```

The antibehavior rule of the frequency divider is similar; the frequency at input **a** is a multiple of that at any output and the window size of measurement is a corresponding fraction.

Behaviorally, two-phase clock generators can be viewed as frequency dividers restricted to a single output that is a two-phase clock; indeed the same physical component may be part of both a frequency divider and of a two-phase clock generator. Their behavior rule says that the output frequency is half that of the input, measured with a window size twice the size of the input:

```
  If  [isa ?c two-phase-clock-generator]
 and  [thru ?l1 ?u1 (mode ?c) normal]
 and  [thru ?l2 ?u2 (fww ?w '(0 1) (ll (in a ?c))) ?f]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
            (fww (* 2 ?w) '(nil t) (cc (out y ?c)))
            (/ ?f 2)]
```

The ordinary behaviors of inverters and toggles in terms of moment-by-moment changes of the logic levels at their inputs and outputs can be described using TINT rules. Rules can also describe their behavior in terms of whether those signals are changing or not and what their frequencies are. Because these behaviors are event preserving, the rules and resulting predictions are strong. Not all behaviors are event-preserving, however; the next three sections present ways of using temporal abstractions in more general situations.

5.6 Reduction

Any function of n inputs with one of its inputs held constant yields a new function of $n - 1$ inputs, and this fact can be used to form a temporally abstracted behavior for a multiple input behavior under the special case of its having one or more constant inputs. The resulting behavior is incomplete, of course, in the sense that it does not cover cases in which the inputs are not constant. It is nevertheless worthwhile because it provides an alternative to the undesirable option of predicting all behavior at a temporally detailed level: weak temporally abstract predictions are better than none.

A simple example is the behavior of a two-input AND gate (denoted `tand2`) under the special case where one of its inputs is the constant signal `(lambda (time) 1)`.

```
tand2 ==
  (lambda (A B) (lambda (time) (logand (A time) (B time))))
```

A straightforward derivation uses the fact that `(logand 1 z) == z` to show that if `X == (lambda (time) 1)` then `(tand2 X Y) == Y`.

The rules for the two-input AND gate (component type `and2`) are shown here; the pattern for OR, NAND, NOR, XOR, and so forth should be relatively clear from these examples. It is tedious but straightforward to write separate rules for gates of the same type but with different arities.

If any input of an AND gate is `0` then the output is `0`:

```
  If  [isa ?x and2]
 and  [thru ?l1 ?u1 (mode ?x) normal]
 and  [thru ?l2 ?u2 (ll (in ?n ?x)) 0]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2) (ll (out y ?x)) 0]
```

Note that one of the considerations in translating the definitions into rules is that the rules should be written in such a way as to reference the minimal sets of facts needed to make their conclusions. Hence sometimes several rules will be used to represent a behavior that was captured with a single function. This is because the troubleshooting engine will examine the dependencies left by the rules to determine which components could have been responsible for observed symptoms. Spurious dependencies make the troubleshooting engine waste effort working on components that could not in fact have caused the symptoms.

The antibehavior rule for the AND-gate says that if the output is 1 then all of the inputs must be 1:

```
  If  [isa ?x and2]
 and  [thru ?l1 ?u1 (mode ?x) normal]
 and  [thru ?l2 ?u2 (ll (out y ?x)) 1
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [has-port ?x (in ?n ?x)]
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
             (ll (in ?n ?x)) 1]
```

Another rule for the AND gate says that with all but one of its inputs held to 1, it acts as a buffer. In the two-input case, this means that as long as input ?n is 1 the output is the same as input (- 1 ?n):

```
  If  [isa ?x and2]
 and  [thru ?l0 ?u0 (mode ?x) normal]
 and  [thru ?l1 ?u1 (ll (in ?n ?x)) 1]
 and  (overlap (?l0 ?u0) (?l1 ?u1))
Then  [tsame (max ?l0 ?l1) (min ?u0 ?u1)
             (ll (in (- 1 ?n) ?x)) (ll (out y ?x))]
```

The latter rule is interesting because the identity between the output and free input will have consequences for any abstraction of either signal, including temporal abstractions. The behavior of the AND gate with all but one of its inputs 1 is one-to-one function that is event preserving just like the inverter. Similarly the behavior of a NAND gate when all but one of its inputs is 1 is just that of an inverter. Hence the temporally abstract version of the NAND gate refers to the changing-wrt abstraction as does the inverter rule:

```
  If  [isa ?x nand2]
 and  [thru ?l1 ?u1 (mode ?x) normal]
 and  [thru ?l2 ?u2 (ll (in ?n ?x)) 1]
 and  [thru ?a ?z GR t]
 and  (<= (max ?l1 ?l2) ?a ?z (min ?u1 ?u2))
Then  [tsame (max ?l1 ?l2) (min ?u1 ?u2)
             (changing-wrt ?a ?z (ll (in (- 1 ?n) ?x)))
             (changing-wrt ?a ?z (ll (out y ?x)))]
```

As with the inverter, any of the abstractions **change**, **stay**, **cycles**, or **fww** could have been chosen, but **changing-wrt** refers to easily observable abstract signals.

The behavior rules of other components in the Console Controller Board are similarly written in a style that makes explicit the event-preserving subsets of their behavior. For example, the behavior of a JK flip-flop with all but its clock input held to 1 becomes **toggle**, which as discussed earlier is partially event preserving. Also, multiplexors are much like buffers, once their **select** input is known. Their principal behavior rule equates the output with whichever input signal is selected:

```
   If  [isa ?m multiplexor]
  and  [thru ?l1 ?u1 (mode ?m) normal]
  and  [thru ?l2 ?u2 (ll (in select ?m)) ?s]
  and  (overlap (?l1 ?u1) (?l2 ?u2))
Then6  [tsame (max ?l1 ?l2) (min ?u1 ?u2)
              (in ?s ?m) (out y ?m)]
```

In general by considering the special case of one or more input signals constant, most behaviors can be reduced to an event-preserving behavior. These restricted temporally abstract behaviors are ubiquitous in the model of the Console Controller Board.

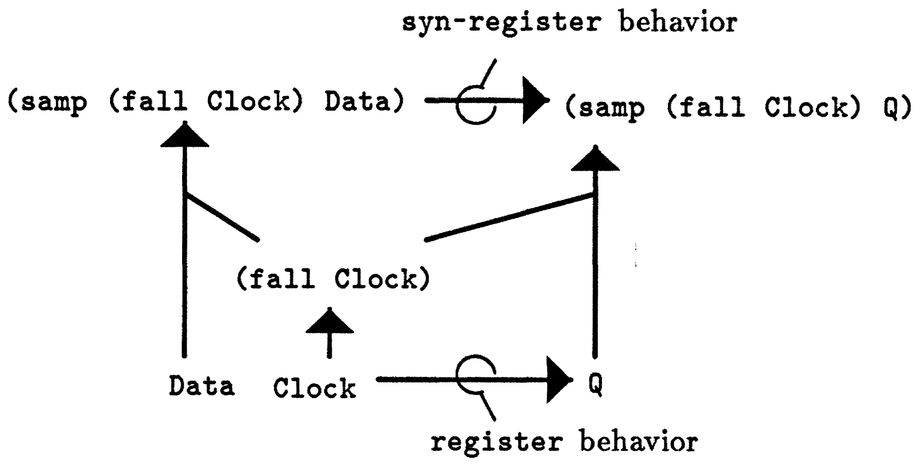
[6] The predicate **tsame** can be used with ports as its third and fourth arguments, not just signals.

5.7 Synchronization

As discussed earlier, temporal abstractions are useful for behaviors whose inputs have known relative timing relationships. An important special case of "known relative timing relationship" occurs when the input signal of some behavior is a clock whose transitions indicate when the other input signals are to be sampled. In that case, the **samp** abstraction can be used to form an abstracted behavior that is more strongly event-preserving than the original. In this section the idea will be used to derive a temporally abstract behavior for a shift register, starting from the behavior of an ordinary register.

The behavior **register** describes the behavior of a falling-edge triggered register; the falling edges of the clock input C capture the data D. **syn-register**, the abstracted version of the **register** behavior, captures the intuition that a register introduces a one-clock delay. Figure 5.9 shows the relationships between the various signals.

Figure 5.9: Register Abstractions



Forming the abstracted version **syn-register** involves several steps. First, the clock signal is abstracted with **fall**. Second, **samp** is used to abstract both the output and D input with respect to the fallings of the clock. Third, the function **synchronous-delay** generalizes **samp** by allowing for arbitrary delays. Finally, the resulting abstracted behavior for the register (**syn-register**) will be easily expressible using **synchronous-delay**.

An example to give some intuition behind these signals is given below. The values at times 2 and 6, when data are latched into the register, are most important. "10" is latched into the register and then "5":

`(syn-del 1 (samp (fall c) d))`	?	?	?	?	?	10	10	10	10	5
`(samp (fall c) d)`	?	10	10	10	10	5	5	5	5	5
`q == (register c d)`	?	10	10	10	10	5	5	5	5	5
`d`	9	10	5	6	5	5	5	4	4	5
`(fall c)`	nil	t	nil	nil	nil	t	nil	nil	nil	t
`c`	1	0	0	1	1	0	0	1	1	0
`time`	0	1	2	3	4	5	6	7	8	9

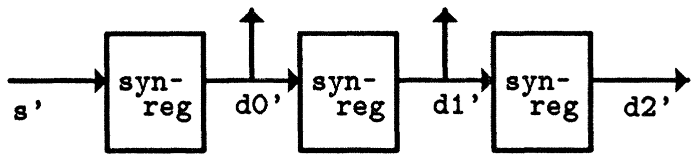
The definition for **synchronous-delay** (abbreviated **syn-del**) resembles that for **samp**, and in fact a delay of 0 is the same as sampling, that is, `(synchronous-delay 0 V S) == (samp V S)`. The abstracted register behavior is then simply "a delay of one clock."

The point of expressing the behavior of the register using the *sampling* abstraction is that the resulting behavior is more strongly event preserving than the lower level **register** behavior. In particular, **register** does not preserve every change in the value of the input signal D; in the example above, d changed from 5 to 6 and back to 5 in between falling edges of the clock, hence those changes were not reflected on the output. The **synchronous-delay** function — and hence the **syn-register** behavior of which it is a special case — is mostly event preserving, even though it is not one-to-one. The result is that the following inequality holds at all **times** for any signals V and S: the number of changes at the output (sampled at V) is within 1 of the number of changes at the input:

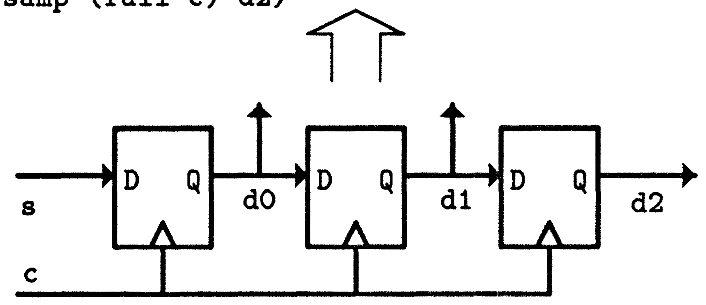
```
         ((count-ww n (change (samp V S))) time)  ≤
((count-ww n (change (syn-register V S))) time)  ≤
  (+ 1 ((count-ww n (change (samp V S))) time))
```

This relationship can now be used to derive the temporally abstract shift register behavior. A shift register configured to convert serial data to parallel can be viewed as a cascade of one-bit registers all sharing a common clock input. Figure 5.10 shows the signals **s'**, d0', d1', and d2'; the components labeled **syn-reg** compute d0' as a function of **s'**, d1' as a function of d0', and so forth:

Figure 5.10: Shift Register as Cascade



```
 s' == (samp (fall c) s)
d0' == (samp (fall c) d0)
d1' == (samp (fall c) d1)
d2' == (samp (fall c) d2)
```



The behavior of a *k*-bit shift register can be expressed with respect to a sampling signal as follows:

```
syn-shift-register ==
  (lambda (k V S) (synchronous-delay k V S))
```

Hence the temporally abstract behavior of a *k*-bit shift register is simply a variation of the inequality shown above for **syn-register**; the number of changes that appear on the synchronous output is within *k* of the number of changes on the synchronous input:

```
        ((count-ww n
            (change (syn-shift-register k V S))) time)  ≤
                ((count-ww n (change (samp V S))) time)  ≤
(+ k ((count-ww
           (change (syn-shift-register k V S))) time))
```

One of the consequences of this relationship is that if the incoming signal to the register has a large enough frequency over a large enough interval, then the output signals will have positive frequencies as well. Suppose it is known that over some time interval, the frequency of changes on a signal was (strictly) bounded below by a positive frequency $\frac{1}{w}$:

$\frac{1}{w} <$ `((fww w '(nil t) (change (samp V S))) time)`

Then the number of changes during any window must be at least 1:

$1 <$ `((count-ww w '(nil t) (change (samp V S))) time)`

Hence for any $k \geq 1$ the number of changes during a window of size $k \times w$ is at least k (provided that $k \times w$ does not get bigger than the interval during which the frequency was known):

$k <$
```
((count-ww (* k w) '(nil t)
    (change (samp V S))) time)
```

Using the previously derived bounds on the number of changes on the outputs of the shift register, the kth output must have at least one change:

$0 <$
```
((count-ww (* k w)
    (change (syn-shift-register k V S))) time)
```

This derivation and its conditions can be summarized into a single relationship. If the incoming signal of the register has a large enough frequency over a large enough interval, the output signals will have positive frequencies as well; the relationship below makes "large enough" precise:

- If `(fww` w `'(nil t) (change V S))` is always $> \frac{1}{w}$ from time l to u, and $k < \frac{(u-l)}{w}$, then from time $l + kw$ to u, `(count-ww (*` k w`) (change (syn-shift-register` k `V S)))` is always > 0.

The Audio Decoder contains two shift registers that accumulate incoming serial data bits. Shift registers used in this fashion are referred to here as *clocked serial accumulators* and the rules of their behavior are based on these relationships. The temporally abstract behavior of a clocked serial accumulator yields the following rules. The first rule infers from the fact that the incoming byte stream is changing that all of the output data bits must be changing:

```
  If  [isa ?csa clocked-serial-accumulator]
 and  [thru ?l1 ?u1 (mode ?csa) normal]
 and  [thru ?l2 ?u2
            (fww ?w '(nil t) (cross 0 (cs (in a ?csa)))) ?f]
 and  (< (/ 1 ?w) ?f)
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [has-port ?csa (out ?k ?csa)]
 and  (< ?k (/ (- (min ?u1 ?u2) (max ?l1 ?l2)) ?w))
 and  [thru ?a ?z GR t]
 and  (<= (+ (max ?l1 ?l2) (* ?k ?w)) ?a ?z (min ?u1 ?u2))
Then  [thru ?z ?z (changing-wrt ?a ?z (ll (out ?k ?csa))) t]
```

The second rule is inherited from an ordinary shift register. All but the last output of a k-bit shift register is an input to the next stage; hence the same relationship holds between output k and output $k+j$ as held between the input and output j; in particular, a changing input implies a changing output, and vice versa. The following rule captures the fact that if output k is observed to be changing then output $k+1$ will, too:

```
  If  [isa ?csa clocked-serial-accumulator]
 and  [thru ?l ?u (mode ?csa) normal]
 and  [thru ?a ?z gr t]
 and  (<= ?l ?a ?z ?u)
 and  [thru ?z ?z (changing-wrt ?a ?z (out ?k ?csa)) t]
 and  [has-port ?csa (out (+ ?k 1) ?csa)]
Then  [thru ?z ?z
            (changing-wrt ?a ?z (out (+ ?k 1) ?csa)) t]
```

The point of using temporal abstractions is to be able to make predictions about component behaviors using simple observations. In this case, there is a strong relationship between the number of changes of value on the input and outputs of the shift register. If more than k changes are observed at the input to the shift register, the temporally abstract behavior can derive bounds on the number of changes that should be observed at its output without requiring clock-by-clock reasoning. What made it possible in this example was the *sampling* abstraction, which allowed us to represent synchronous signals and thereby describe the behavior of a register as a component that introduces a delay between signals.

5.8 Encapsulation

Sequential circuits are more difficult to reason about than combinational circuits. In general, predicting the response to a particular sequence of stimuli may require explicitly representing every intervening state change. The more complex the circuit behavior — that is, the more distinguishable states that the circuit can be in — the greater the need for temporal abstractions to simplify that reasoning. Up to this point in the discussion, examples of temporally abstract behaviors have all been either combinational circuits or very simple sequential circuits such as shift registers. This section uses the previously discussed abstractions and abstraction techniques to develop examples of temporally abstract behaviors for more complex sequential circuits. The ideas being illustrated are simple:

1. The behavior of a group of components appearing in a loop can be expressed as the composition of the component behaviors by introducing a new signal that represents the state of the aggregate component. This encapsulation alone does not usually simplify reasoning about the behavior of the loop.

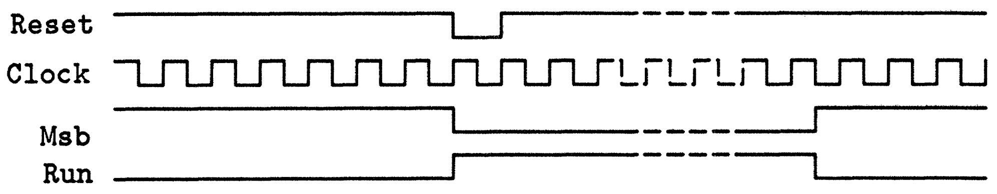
2. The goal of abstracting the behavior of a sequential circuit is to collapse together equivalent states in its state diagram — ideally down to a single state so that the output of the circuit can be expressed directly in terms of its inputs without the intervening "state" signal.

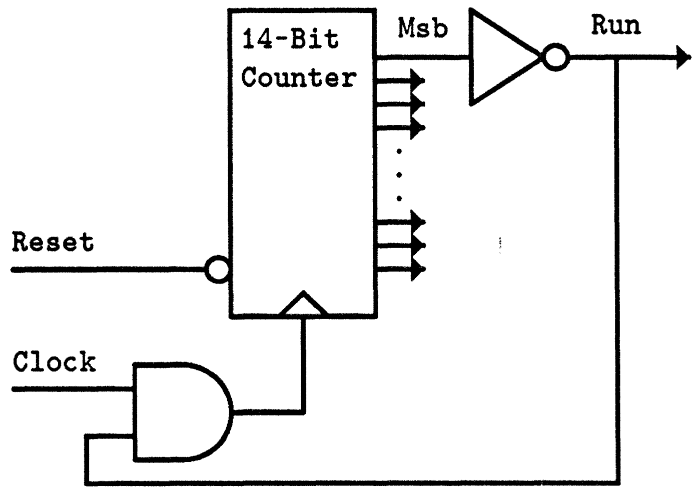
3. If the behavior of a sequential device involves performing computations that are similar to counting, sampling, recognizing sequences, and so forth, then a powerful way to simplify its behavior (that is, reduce the number of distinguishable states) is to describe its inputs in terms of corresponding temporal abstractions such as `count`, `sample`, and `sequence`.

5.8.1 The Reset Hold Counter

The Reset Hold Counter circuit (Figure 5.11) from the Console Controller Board is a simple example that illustrates the role of loop encapsulation in deriving temporally abstract behaviors. When the `Reset` signal is asserted and the clock signal Clock is running at k Hz, the Run signal is asserted for at least $\frac{2^{13}}{k}$ seconds.

Figure 5.11: Reset Hold Counter

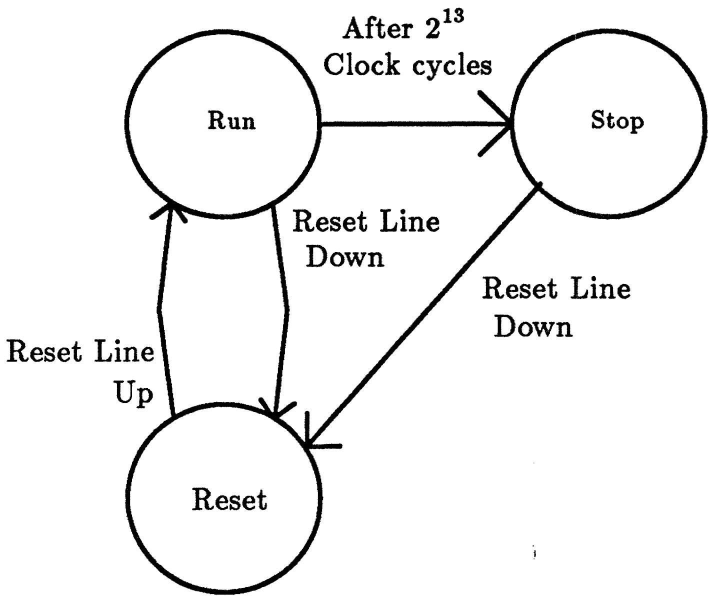




This circuit, containing a 14-bit counter, has at least 2^{14} distinguishable states, but by using the temporal abstractions it is possible to describe its behavior using only three states. The intuition behind this is that if the Clock input is known to be periodic, and it is known how long it has been since the counter has been reset, then the state of the counter (and hence of the circuit as a whole) is computable from the product of the clock frequency and the length of time the Reset signal has been 1. The temporally abstract behavior rh is derived at length in Appendix C.

This behavior can be described as a three-state automaton (Figure 5.12). The automaton has one of each of three general kinds of transition conditions: (i) transitions out of certain states caused by input events; (ii) transitions that occur no matter what the previous state was; (iii) transitions arising from being in a given state for a certain amount of time. The interaction between these three kinds of transitions shows up as somewhat complex persistence rules. Complex as it is, the encapsulation of the entire circuit along with the *frequency* temporal abstraction allows the resulting behavior to be quite simple relative to the counter that underlies it.

Figure 5.12: Reset Hold Counter Three State Automaton



The first transition rule says that when the **reset** input is **1**, the component goes into the **Reset** state. It is not necessary to know the previous state nor the previous value of the input, so this rule is simpler than most transition rules:

```
  If  [isa ?r reset-hold]
 and  [thru ?l1 ?u1 (mode ?r) normal]
 and  [thru ?l2 ?u2 (l1 (in reset ?r)) 1]
 and  (<= (+ δ (max ?l1 ?l2)) (min ?u1 ?u2))
Then  [thru (+ δ (max ?l1 ?l2)) (+ δ (max ?l1 ?l2))
            (state ?r) Reset]
```

The persistence rule associated with the Reset state says that ?R stays there as long as there are no changes from 1 to 0 on the reset input:

```
  If  [isa ?r reset-hold]
 and  [thru ?l1 ?u1 (mode ?r) normal]
 and  [thru ?l2 ?u2 (event 1 0 (l1 (in reset ?r))) nil]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (state ?r) Reset]
 and  (<= (max ?l1 ?l2) ?u2 (min ?u1 ?u2))
Then  [thru (max ?l1 ?l2 ?l3) (+ δ (min ?u1 ?u2))
            (state ?r) Reset]
```

From the Reset state, a change from 1 to 0 on the reset input causes a transition to the Run state. Typical of most transition rules, the conclusion is only warranted at the single moment after the input changed:

```
  If  [isa ?r reset-hold]
 and  [thru ?l1 ?u1 (mode ?r) normal]
 and  [thru ?l2 ?u2 (event 1 0 (l1 (in reset ?r))) t]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (state ?r) Reset]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
Then  [thru (+ δ ?u1) (+ δ ?u1) (state ?r) Run]
```

A separate persistence rule extends the Run state until the reset input is asserted, or until 2^{13} clock cycles have elapsed. Given a frequency of ?f, 2^{13} clock cycles elapse in $2^{13} \times \frac{1}{?f}$ seconds:

```
  If   [isa ?r reset-hold]
 and   [thru ?l1 ?u1 (mode ?r) normal]
 and   [thru ?l2 ?u2 (event 0 1 (ll (in reset ?r))) nil]
 and   (overlap (?l1 ?u1) (?l2 ?u2))
 and   [thru ?l3 ?u3 (fww ?w ?cyc (ll (in clk ?r))) ?f]
 and   (> ?f 0)
 and   (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
 and   [thru ?l4 ?u4 (event :any run (state ?r))]
 and   (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3) (?l4 ?u4))
Then   [thru (max ?l1 ?l2 ?l3 ?l4)
             (+ δ (min (+ (max ?l1 ?l2 ?l3 ?4) $2^{13} \times \frac{1}{?f}$)
                          (min ?u1 ?u2 ?u3)))
             (state ?r) Run]
```

A transition rule for the Run state makes the transition to the Stop state happen when enough clock cycles have passed (that is, 2^{13} cycles):

```
  If   [isa ?r reset-hold]
 and   [thru ?l1 ?u1 (mode ?r) normal]
 and   [thru ?l2 ?u2 (state ?r) Run]
 and   (overlap (?l1 ?u1) (?l2 ?u2))
 and   [thru ?l3 ?u3 (fww ?w ?cyc (ll (in clk ?r))) ?f]
 and   (<= (* (/ 1 ?f) $2^{13}$) (- ?u2 ?l2))
Then   [thru (+ δ ?u2) (+ δ ?u2) (state ?r) Stop]
```

Finally, the Stop state persists so long as no 0 to 1 changes occur on the reset input:

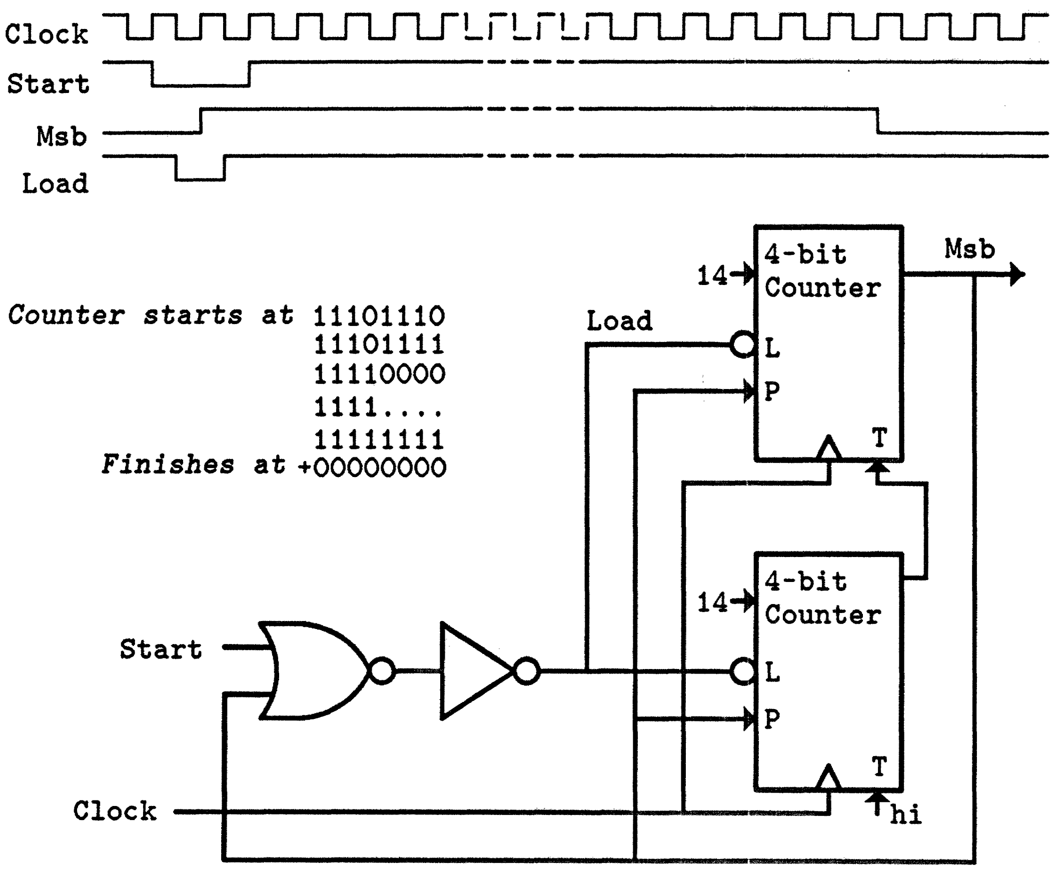
```
  If   [isa ?r reset-hold]
 and   [thru ?l1 ?u1 (mode ?r) normal]
 and   [thru ?l2 ?u2 (event 0 1 (ll (in reset ?r))) nil]
 and   (overlap (?l1 ?u1) (?l2 ?u2))
 and   [thru ?l3 ?u3 (state ?r) Stop]
 and   (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
Then   [thru (max ?l1 ?l2 ?l3) (+ δ (min ?u1 ?u2))
             (state ?r) Stop]
```

The behavior of the Reset Hold Counter can thus be expressed compactly by representing the state of the counter implicitly with the *frequency* and *duration* temporal abstractions.

5.8.2 The Audio Counter

The Audio Counter (Figure 5.13) bears obvious similarities to the Reset Hold Counter discussed above, but it has subtle differences that lead to a different temporally abstract behavior. The relevant temporal abstraction is the samp abstraction encountered earlier. Using this abstraction the temporally abstract behavior of the encapsulated Audio Counter will resemble that of a frequency divider.

Figure 5.13: Audio Counter



While the Reset input of the Reset Hold Counter starts the counter back at 0 whenever asserted, in the Audio Counter only the first 1-to-0 transition of the `Start` signal matters. Eighteen clock cycles must pass before the zero state can be reached again: while counting, it is insensitive to the `Start` signal.

Some temporal abstractions that applied to earlier examples can be applied to the behavior of this circuit; however, the assumptions on which they depend are violated by the normal usage of the circuit and so the resulting temporally abstract behaviors have little predictive force. For example, while the signal `Msb` is a constant `1`, the Audio Counter forms a frequency divider with respect to the `Clock` input; however, the clocks come in bursts of eighteen and normally the `Start` line goes low at least once per burst — the "frequencies" are thus defined over so few cycles as to be useless. For another example, the "counting" behavior of the Audio Counter can be captured by the product of a frequency and a duration, but only during the bursts of eighteen clock cycles and hence this is similarly useless.

Appendix D shows the derivation of a behavior that is event-preserving: n falling edges on the `Start` signal sampled with respect to falling edges of the `Clock` will result in somewhere between $\lfloor\frac{n}{18}\rfloor$ and n falling edges on `Msb`. Thus the number of falls on `Msb` (measured with respect to rising edges of `Clock`) is bounded as follows:

```
((count-ww
    n (fall (samp (rise Clock) Start))) time)  ≥
((count-ww
      n (fall (samp (rise Clock) Msb))) time)  ≥
(floor
  ((count-ww
    n (fall (samp (rise Clock) Start))) time)
  18)
```

A similar inequality was derived earlier for the shift register, with the consequence that a relationship could be defined between the frequency at the input and at the output. A similar derivation for the eighteen-counter results in a similar relationship. If the incoming signal of the counter has a large enough frequency over a large enough interval, the output signals will have positive frequencies as well; the relationship below makes "large enough" precise:

- If `(fww` w `'(nil t) (change V S))` is always $> \frac{1}{w}$ from time l to u, and $18 < \frac{(u-l)}{w}$, then from time $l + 18w$ to u, `(count-ww (*` 18 w`) (change (eighteen-counter V S)))` is always > 0.

The relevant behavior rule thus looks very similar to the rule for the accumulator; the difference is that the conditions under which it can be deduced that a changing input will result in a changing output are more restricted than for the accumulator:

```
  If  [isa ?csb clocked-serial-burst-detector]
 and  [thru ?l1 ?u1 (mode ?csb) normal]
 and  [thru ?l2 ?u2
            (fww ?w '(nil t) (cross 0 (cs (in a ?csb)))) ?f]
 and  (< (/ 1 ?w) ?f)
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  (< 18 (/ (- (min ?u1 ?u2) (max ?l1 ?l2)) ?w))
 and  [thru ?a ?z GR t]
 and  (<= (+ (max ?l1 ?l2) (* 18 ?w)) ?a ?z (min ?u1 ?u2))
Then  [thru ?z ?z (changing-wrt ?a ?z (l1 (out y ?csb))) t]
```

This temporally abstract behavior rule is useful because it uses simple observations about signals to yield other easily observed predictions.

5.8.3 Microprocessors

The behavior of a microprocessor can in principle be represented as an enormous finite state automaton. However, its behavior can be represented in a temporally abstract way by characterizing its behavior in just two states: `Stop` and `Run`. The Console Controller Board contains two eight-bit microprocessors, an Intel 8035 and an Intel 8741. These microprocessors run instructions only when their incoming clocks are valid two-phase clocks of no more than 5 Mhz. The abstraction `two-phase-clock` maps a pair of {`1`, `0`} signals to {`t`, `nil`}, where `t` marks the end of a two-phase clock cycle. To be in the `Run` state the processors must have their `reset` input unasserted and the incoming clock signal be a valid two-phase clock with frequency less than 5 Mhz:

$$0 < \begin{array}{l}\texttt{((fww n '(nil t)} \\ \quad\texttt{(two-phase-clock C5MhzH C5MhzL)) time)}\end{array} \leq 5 * 10^6$$

When not running they are in the Stop state and their outputs are idle. In the Run state some of their outputs are periodic. For example, In the Run state of the I8035 with a 5 Mhz clock, the PSEN output runs at 50 Khz and the ALE output at 300 Khz; these frequencies are asserted by a rule using window sizes corresponding to a thousand cycles of each signal:

```
  If   [isa ?i i8035]
 and   [thru ?l ?u (state ?i) Run]
Then   [thru ?l ?u (fww 1000 × (1 ÷ 300Khz)
                      '(0 1) (ll (out psen ?i))) 300Khz]
 and   [thru ?l ?u (fww 1000 × (1 ÷ 50Khz)
                      '(0 1) (ll (out ale ?i))) 50Khz]
```

Similarly, in its Run state the I8741 provides clocks and initialization signals to the keyboard and keypad at frequencies in the neighborhood of 20 Khz.

5.8.4 Abstract Buffers

The behavior of a single-input single-output buffer is the identity function: its output at each moment is the same as its input. Since the identity function is one-to-one, buffers are event preserving and lend themselves to temporally abstract behavior descriptions. There are only a few buffers *per se* in a typical digital circuit. On the other hand, digital circuits often have substantial amounts of circuitry devoted to doing information-preserving transformations of data from one encoding to another — from serial to parallel, for example. At the right level of temporal abstraction, modulo the nuances of the different data formats, many seemingly complex circuits are really "buffers" in this broader sense. Buffers thus appear in the Console Controller Board at various levels of temporal abstraction. In the Audio Counter the Manchester-to-serial converter, for example, is just a buffer when viewed with respect to incoming (Manchester encoded) and outgoing (encoded serially with a clock) signals:

```
  If   [isa ?m manchester-to-serial]
 and   [thru ?l ?u (mode ?m) normal]
Then   [tsame ?l ?u (manchester (in a ?m)) (cs (out y ?m))]
```

Similarly, the serial-to-parallel converter can be viewed as a buffer between byte streams encoded synchronously and serially (`cs`) and in parallel (`cp`):

```
  If  [isa ?s serial-to-parallel]
 and  [thru ?l ?u (mode ?s) normal]
Then  [tsame ?l ?u (cs (in data ?s)) (cp (out y ?s))]
```

The behavior rules for buffer-like behaviors all deduce a `tsame` relation between their inputs and outputs.

5.8.5 Programmed Microprocessors

Encapsulation and temporal abstraction can be applied to circuits containing microprocessors. In doing so, the resulting behaviors collapse large state transition diagrams into tiny ones and sacrifice a great deal of precision. They are useful for troubleshooting because they allow predictions to be made efficiently about temporally coarse features of signals. The behavior model for the 8741 processor on the Console Controller Board, for example, predicts little more than that if the processor is running, rolling the mouse around will cause it to assert one of its outputs several hundred times a second. Although very coarse, it is useful because (i) it is easy to distinguish between that output being idle and being very active (ii) a significant fraction of faults in the processor would cause that output to be idle, and (iii) it is more efficient than reasoning about hundreds of identical events individually. The examples will be presented by encapsulating the component behaviors in bottom-up fashion, eventually constructing a behavior for a group of several chips including two microprocessors.

(The material in the remainder of this section involves many details specific to the Console Controller Board; readers pressed for time may wish to skip forward to Page 163.)

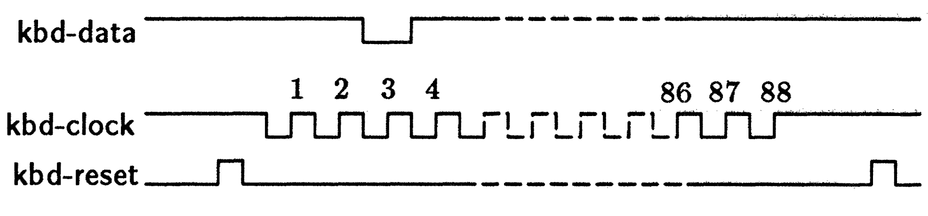
The temporally abstract behavior of U, the Input Processor, was used in the Input Encoder troubleshooting examples (Figure 3.10 on Page 58 shows the functional organization of the Input Encoder). U consists of the Intel 8741 microprocessor mentioned above, along with the onboard PROM that stores its control program. Most of the behavior of U is simple enough to represent usefully with temporal abstractions. With the right temporal abstractions and assumptions about its incoming signals, its behavior can be expressed as a combinational function of its inputs. The essential abstractions making this possible are as follows:

- For troubleshooting purposes, the most important properties of the incoming keyboard and mouse data signals can be concisely expressed in terms of changes and rates of change.
- Although U sends all its output packets over a common eight-bit bus, the rate at which different types of packets are sent is substantially different, and this can be taken advantage of in representing its temporally abstract behavior.

The Keyboard and Keypad inputs are encoded serially and synchronously, while the behavior of the Console Controller Board refers to changes in the state of individual keys. Temporal abstractions are needed to map from the low level encoding up to the level of changes in key positions. The Keyboard signal is taken as the specific example; the Keypad signal is treated analogously.

The full state of the keyboard — the position of every key — is transmitted repeatedly to the Input Processor approximately a thousand times a second. There are three digital signals that accomplish this, kbd-reset, kbd-clock, and kbd-data. Figure 5.14 shows an example: the kbd-reset signal is asserted to indicate that a new scan of the keyboard is beginning, kbd-clock has one rising edge for each of 88 keys, and kbd-data is 0 wherever the corresponding key is pressed, in this case the key in the third position on the keyboard. While all the keys are up the signal kbd-data is a constant 1.

Figure 5.14: The Third Key is Pressed



The temporally abstracted signal kbd-state represents the accumulated bits in each previous sequence of 88 clock cycles. The remaining abstraction needed is to represent the signal in terms of changes of the state of the keyboard, that is, of changes in the position of the keys. The signal **kbd-events** represents that abstraction.

kbd-events	nil ...	(Abort Down)	nil
kbd-state	0 ...	512	0
`time`	0 ...	1000	1001 ...

These abstractions map from underlying serial signals up to a vocabulary of events on individual keys. No further temporal abstractions are needed for representing keys, since the rate at which keys can change is low enough to be easily observable.

Like the keyboard inputs, the inputs from the mouse are encoded in a way that is too low-level to be useful for troubleshooting; all that really needs to be represented is whether the mouse is traveling in the x and y dimensions. Again, temporal abstractions can map from the level of implementation up to rates of travel. The movement of the mouse along the x axis is represented using a 2-bit gray code on the (misleadingly named) signals **mouse-left** and **mouse-right**. Each move by $\frac{1}{100}$ inch in the positive direction results in one of the events `(0 0)` $\rightarrow$ `(0 1)` $\rightarrow$ `(1 1)` $\rightarrow$ `(1 0)` $\rightarrow$ `(0 0)`; the reverse for the negative direction. Hence the net travel (*not* the net change in position) during an interval n yields the number of events. The temporally abstract signals **mouse-dx** and **mouse-dy** are defined with an observation window size of one second (since the mouse travels at up to 10 inches per second, hence there are 1000 events per second, hence a one-second window is 1000 times the typical period).

The behavior of **U** can now be expressed in terms of the signals just described, namely, the temporally abstract clock, keyboard, and mouse inputs, along with the **Reset** input. While the **Reset** line is asserted the outputs of **U** are inactive. While the clock is running (that is, while the signal `(two-phase-clock` **C5MhzH C5MhzL**`)` has frequency 5 Mhz) the **I8741** waits for events indicating mouse motions and keystrokes, and when such a event occurs it asserts the interrupt line **int**, causing an interrupt cycle and ultimately resulting in the transfer of a packet to **C**. The behavior of **U** thus merges the various incoming events into a single outgoing stream of packets. The output signal **packets** is defined so that it is `nil` everywhere except when a packet is being transmitted, for example, `'(Local Down)` to represent the "local" key being pressed, '(Mouse Right) to indicate that the mouse has moved $\frac{1}{100}$ inch to the right, and so forth.

The temporal scale at which mouse events and keyboard events occur and their effects on the behavior of the Input Encoder are substantially different. Mouse motion, for example, never changes the state of the Input Encoder, while events of the "local" key change the behavior of Input Encoder dramatically. Furthermore, it is rarely the case that the mouse is rolled around at the same time as the keyboard is being typed at — or at least this can be guaranteed while troubleshooting. As a consequence, it is useful to define the behavior of U under these different conditions at two different temporal resolutions:

1. While the mouse is inactive, packets essentially merges the keyboard and keypad events, with int being asserted once per packet.

2. While the keyboard and keypad are inactive, (tsign (count-ww n packets)) is just the (qualitative) sum of the mouse-dx and mouse-dy inputs, with (tsign (count-ww n (fall int))) having the same value.

Under conditions 1 and 2, U preserves events on the keyboard and mouse inputs respectively; the different rates at which such events occur means that different temporal abstractions are appropriate for representing the resulting behavior.

The Input Processor U, like the I8741 that comprises it, has a Stop and a Run state. The difference between the Input Processor and the I8741 is the level of abstraction of their inputs and outputs. The inputs of U are the temporally abstract keyboard, keypad, and mouse inputs. The incoming kbd-state signal that transmits the state of the keyboard appears at (ks (in kbd U)), and for the keypad at (ks (in kpd U)). The keyboard-events abstraction applied to these signals yield, respectively, the inputs (kt (in kbd U)) and (kt (in kpd U)). The signals mouse-dx and mouse-dy transmitting the direction of mouse motion appears at (mmx (in mouse U)) (along the x axis) and (mmy (in mouse U)) (along the y axis). The output of U is the interrupt signal INT. The salient rules governing the behavior of the Input Processor are given below.

While in the Stop state (that is, while the reset line is asserted), the INT output signal is a constant 1:

```
    If  [isa ?i input-processor]
   and  [thru ?l ?u (state ?i) Stop]
  Then  [thru ?l ?u (ll (out int ?i)) 1]
```

While the mouse inputs are idle, each incoming keyboard or keypad event results in the interrupt line being held low:

```
    If  [isa ?i input-processor]
   and  [thru ?l1 ?u1 (state ?i) Run]
   and  [thru ?l2 ?u2 (mmx (in mouse ?i)) 0]
   and  (overlap (?l1 ?u1) (?l2 ?u2))
   and  [thru ?l3 ?u3 (mmy (in mouse ?i)) 0]
   and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
   and  [thru ?l4 ?u4 (kt (in kbd ?i)) ?kbd]
   and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3) (?l4 ?u4))
   and  [thru ?l5 ?u5 (kt (in kpd ?i)) ?kpd]
   and  (overlap (?l1 ?u1) (?l2 ?u2)
                 (?l3 ?u3) (?l4 ?u4) (?l5 ?u5))
  Then  [thru (max ?l1 ?l2 ?l3 ?l4 ?l5)
              (min ?u1 ?u2 ?u3 ?u4 ?u5)
              (ll (out int ?i))
              (if (or ?kbd ?kpd) 1 0)]
```

As long as no keyboard or keypad events occur, changes on the interrupt line occur only when there is motion on the mouse inputs:

```
  If  [isa ?i input-processor]
 and  [thru ?l1 ?u1 (state ?i) Run]
 and  [thru ?l2 ?u2 (kt (in kbd ?i)) nil]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (kt (in kpd ?i)) nil]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
 and  [thru ?l4 ?u4 (mmx (in mouse ?i)) ?mmx]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3) (?l4 ?u4))
 and  [thru ?l5 ?u5 (mmy (in mouse ?i)) ?mmy]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3)
               (?l4 ?u4) (?l5 ?u5))
 and  [thru ?a ?z GR t]
 and  (<= (max ?l1 ?l2 ?l3 ?l4 ?l5) ?a
          ?z (min ?u1 ?u2 ?u3 ?u4 ?u5))
Then  [thru (max ?l1 ?l2 ?l3 ?l4 ?l5)
            (min ?u1 ?u2 ?u3 ?u4 ?u5)
            (changing-wrt ?a ?z (l1 (out int ?i)))
            (eql '+ (qplus ?mmx ?mmy))]
```

Finally, the Input Processor has an antibehavior rule that infers that the Reset line must be 0 if there was a keyboard event but the interrupt line was never asserted. This is a compression of a more complex line of reasoning that would infer that it must have been in the Stop state:

```
  If  [isa ?i input-processor]
 and  [thru ?l1 ?u1 (mode ?i) normal]
 and  [thru ?l2 ?u2 (l1 (out int ?i)) 1]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (kt (in kbd ?i)) ?event]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
 and  (not (null ?event))
Then  [thru (max ?l1 ?l2 ?l3) (min ?u1 ?u2 ?u3)
            (l1 (in reset ?i)) 0]
```

As noted at the beginning of this subsection, the temporally abstract behavior of U is a combinational function of its inputs. This was made possible by temporal abstractions that (i) represented the incoming clocks in terms of their frequency and relative phase (ii) represented the other inputs in terms of their events and rate of events, and (iii) matched the rate at which certain events occur. The resulting behavior exposes the simple, important, event-preserving relationship between keystrokes, mouse motions, and activity on the interrupt signal int.

The component C treated as a "black box" in the Input Encoder troubleshooting examples has a similarly abstract behavior description. C is actually the culmination of three intermediate levels of structural composition and behavioral abstraction. This behavior will be developed starting at the lowest level. The first level of composition contains a loop that involves an Intel 8035 microprocessor, a PROM, and two ancillary chips; the result of that composition will be called P. There are three essential abstraction steps:

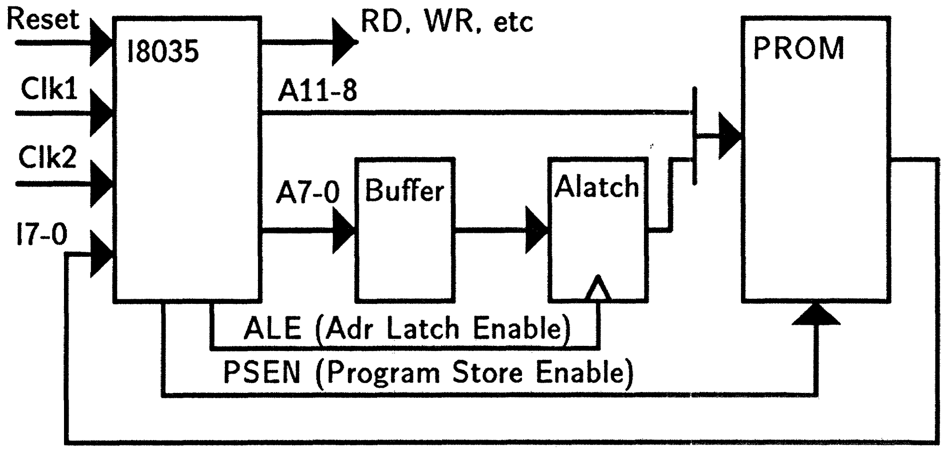
1. The microprocessor communicates via a bidirectional bus, but this complicates behavior descriptions; hence a distinction is made between the incoming and outgoing signals of the microprocessor, sent along the same bus at different times.

2. At the temporal scale of individual instructions, each address that the microprocessor presents to the PROM depends on what the previously returned instruction was. However, some of the outputs of the microprocessor do not depend on the instructions being executed, and this fact can be used to form useful temporal abstractions of the microprocessor behavior.

3. Temporal abstractions can simplify the composed behavior of the microprocessor and PROM down to only four states. This drastic simplification is possible because most of the time the Console Controller is merely buffering incoming data from the keyboard and mouse.

Figure 5.15 shows the microprocessor I8035 and the components used to present instruction addresses to the PROM (the original circuit schematic is part of Page 60). The eight-bit bidirectional bus connected to the microprocessor ports AD7-0 has been divided into an outgoing "address" signal A7-0 and an incoming "instruction" signal I7-0. The signals are valid at different times in the basic instruction cycle of the I8035, and the abstract signals shown are the results of a *sampling* abstraction with respect to the clocks Clk1 and Clk2.

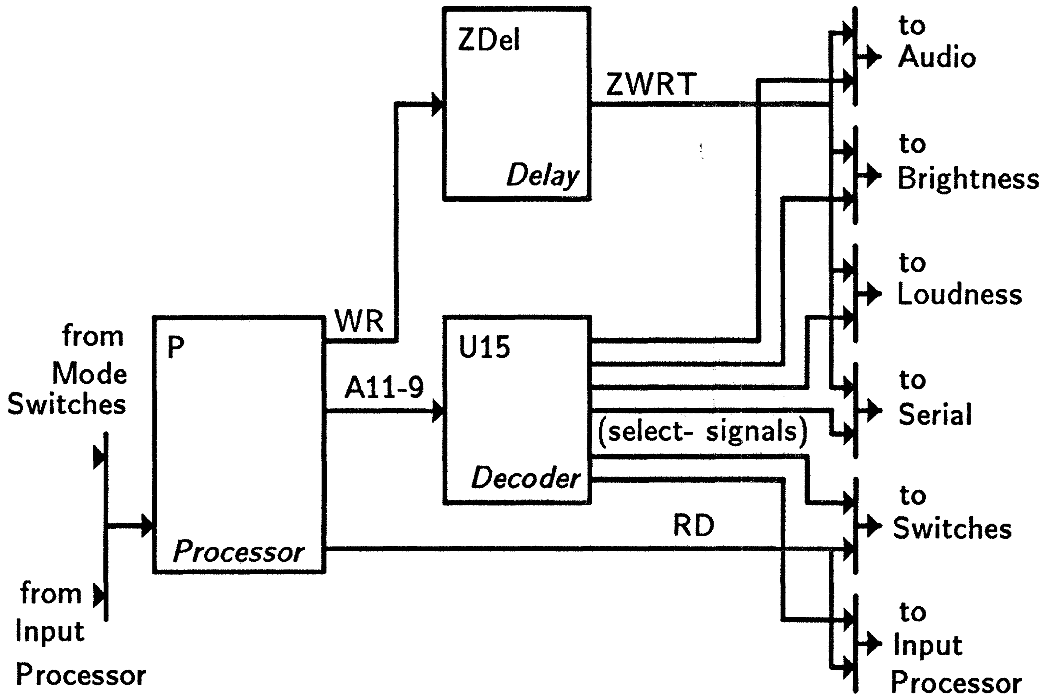
Figure 5.15: Functional Organization of Console Controller



The structural composition of the I8035, PROM, and ancillary components forms the component P, whose behavior is just that resulting from the I8035 executing the program stored in the PROM. The stored console control program implements an idle loop that responds to interrupts from the Input Processor by reading a packet and (usually) sending it on to the host. Some sequences of keystroke packets are not sent on, but are intercepted and cause the program to perform operations local to the console, such as changing the brightness of the screen. Portions of the behavior of P can be described in a temporally abstract way. For example, the eight bits of the main bus over which the addresses and instructions are transmitted should never be flat for more than a few clock cycles; similarly, during the idling loop of the program the RD and WR signals are asserted periodically. For these signals, `(tsign (fww n '(0 1) ...))` is + while C is running. Although P has less complex behavior than the microprocessor — it has fewer distinct states — aside from these few signals it still does not lend itself to temporal abstraction; its interactions with U, for example, must be reasoned about at the level of individual instructions.

P communicates with several slave components via a bidirectional bus, but since most of these communications are one-way, it is useful to represent the paths between the processor and each slave as a separate signal. This abstraction is represented as a second level of composition that forms the component B. B is a composition of P along with the addressing and timing circuitry that mediates these communications (Figure 5.16). The Audio Decoder, Brightness and Loudness registers, and the Serial Encoder are all write-only; the mode switches and Input Processor are read-only.

Figure 5.16: Components of B



Each of the input and output signals of P is a temporal slice of the bidirectional bus that P communicates over. That is, it is the result of *sampling* the bus at particular moments. To be specific, the value that the abstract signal carries is the value being sent at the moment to the given destination, and is otherwise `nil`. An example in which the value "20" is being written to the brightness register is represented as:

to-brightness	nil	nil	nil	20	nil	nil
zwrt	1	0	0	1	0	0
select-brightness	0	0	1	1	1	0
bus	?	20	20	20	20	?
time	0	1	2	3	4	5

The signals **to-audio**, **to-loudness**, and **from-switches** are eight-bit integers like **to-brightness**; the signals **from-input-processor** and **to-serial** carry "packets," to be defined below.

The third and final level of composition that forms C is a loop encapsulation that combines B with the mode switches that control certain minor aspects of its behavior. The switches are read from repeatedly during the idle loop of P, hence this encapsulation results in some simplification of the overall behavior.

The interrupt-response cycle that accomplishes the transmission of packets from component U to component C forms a loop (Figure 5.17). U interrupts C by asserting its **int** signal, C responds by asserting RD; two eight-bit words forming a packet are then transmitted from U to C as the signal **packets**. The combined behavior of these two components is complex, and there may be hundreds of interrupt cycles for a single mouse motion. Encapsulating the loop as component E and using temporal abstractions can reduce the behavioral complexity to manageable proportions. The temporal abstractions that apply to U and C individually have been discussed earlier; here only the combined behavior of the two is considered. That behavior lends itself to temporal abstractions that reduce it to only four distinct states (Figure 5.18). The four-state diagram arises as a consequence of the following observations about U and about the instruction-level behavior of C:

1. The interrupt-cycle interaction between U and C is fully encapsulated within E. Furthermore, it was shown that the behavior of U is event-preserving and state-free under the right temporal abstractions. Hence the behavior of E is mostly dependent on the state-transition behavior of C.

2. Like many state machines, C has a "reset" input that puts it into `Reset` state in which it does nothing. However, it also requires an initialization procedure of about a hundred instructions before actually responding

Figure 5.17: Components U and C Together form Component E

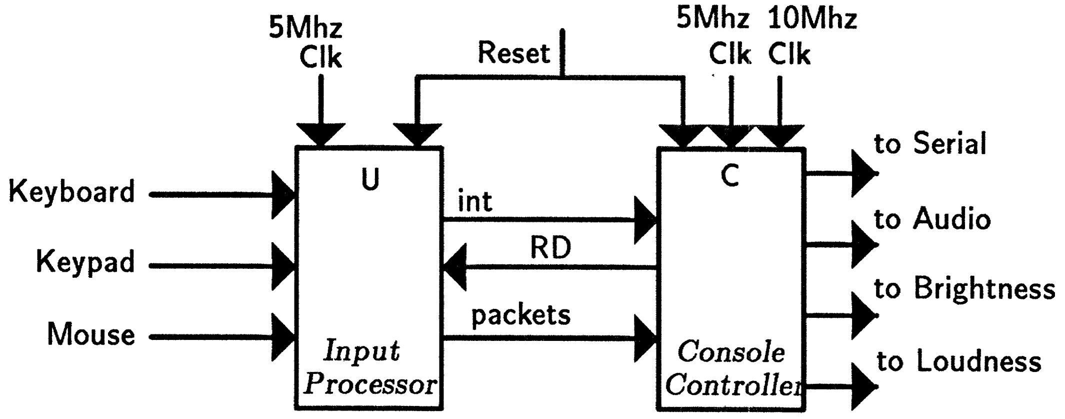
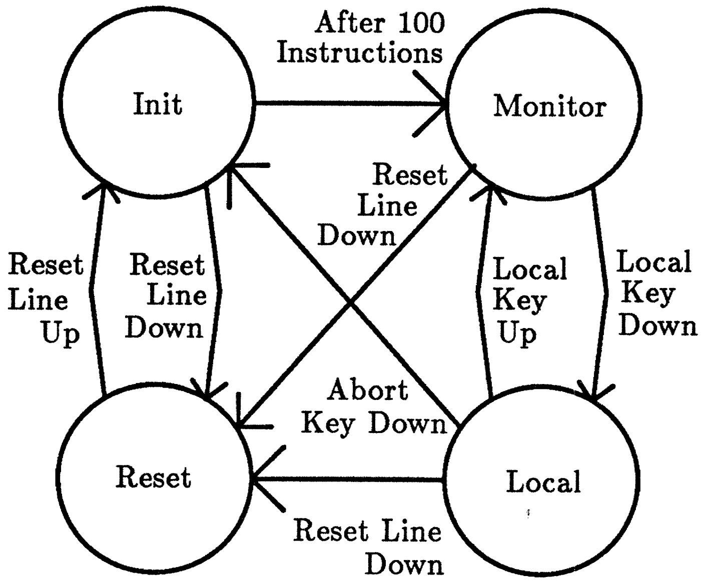


Figure 5.18: State Diagram of E



to any inputs. This instruction sequence can be treated as a separate `Init` state.

3. C is fundamentally interrupt-driven; after being initialized, most of the time it is waiting idly for interrupts. More important, after most interrupts it returns to the same state as it had before. This suggests that most of its behavior can be captured as a function of the most recent input event, without any reference to earlier events. This is its behavior in the `Monitor` state.

4. What behavior of C cannot be captured as a function of the most recent event is capturable in terms of the *counting* and *duration* abstractions. Such behaviors all occur in the `Local` state.

The behavior of E in each of its four states can now be discussed in more detail.

While in the `Reset` state all of the outputs of E are held constant; to-brightness, for example, is `nil`. E remains in this state as long as the reset input is asserted (0).

The `Init` state is entered when the reset input becomes 1 and the frequency of the two-phase clock input is greater than 0 and less than 5 Mhz. E remains in the `Init` state for 600 cycles of the two-phase clock input, since there are about a hundred instructions and each requires six clock cycles. There are a few output operations performed during the `Init` state: the eight-bit brightness and loudness registers are set to an average value, and an initialization sequence of some 40 bytes is sent to the Serial Encoder. Thus, for example, the output signal to-brightness transmits the value 128 during the `Init` state, while the to-serial signal transmits the special token `'init` representing the initialization sequence of the Serial Encoder.

In the `Monitor` state, E behaves very much as U does: events on the incoming keyboard and mouse inputs are converted to packets and sent to the output, in this case the `to-serial` output. The sole exception is the event `'(Local down)`, which is not transmitted but rather causes a state transition to the `Local` state.

The complex behavior of E occurs during the `Local` state. Events on the mouse are sent unchanged to to-Serial as in the `Monitor` state, but some keystrokes cause activity on the to-Audio, to-Brightness, and to-Loudness output signals.

The G key is used to produce a tone on the speaker. While the G key is held down[7] the to-audio signal carries a repeating sequence of integers forming a sinusoidal signal of frequency 1 Khz and of amplitude 128. For troubleshooting, the important properties of this signal are crossings of its midpoint value, both in its first and second derivatives (as introduced in the Audio Decoder troubleshooting examples). The two temporally abstract signals shown below both have the value "1 Khz" while G is pressed, and are 0 otherwise:

```
(fww 1sec '(nil t)
  (cross 127 (samp to-Audio to-Audio)))

(fww 1sec '(nil t)
  (cross 0 (dt (samp to-Audio to-Audio))))
```

The B key is used to brighten the screen continuously, from 0 up to a maximum brightness of 255. While the B key is held down, the to-Brightness signal increases at a rate of 3msec per step until it reaches 255. Conversely, the D key dims it. Just as the "counting" behavior of the Reset Hold Counter could be expressed in terms of the duration abstraction, similarly the to-brightness output can be expressed in terms of the lengths of time the B and D keys have been pressed.

The L and Q keys work analagously to the B and D keys, sending to the to-loudness signal and making subsequent audio signals louder (L) and quieter (Q).

The six rules for the Reset Hold Counter together implemented the three-state automaton shown in Figure 5.12 (Page 142) — not an unusual ratio of rules-to-states, since a typical state diagram will require roughly one transition rule per arc and one persistence rule per state. Writing the rules is sufficiently tedious that a prerequisite to managing a large finite-state diagram would be to develop machinery for automatically translating the graph into rules. Because that has yet not been done, the temporally abstract behavior of the component E, with its four states and eight arcs, is the largest behavior implemented to date. The transition and persistence rules for E and its subcomponents are sufficiently similar to those for the Reset Hold Counter that they will not be duplicated here.

[7]That is, while `(aref (kbd-state time) (key->pos 'G))` is 0.

This completes the temporally abstract behavior of E. The important point is its simplicity — perhaps not simple in comparison with the simplicity of the behavior of a boolean gate, but vastly simpler than the behaviors of the underlying microprocessors. The simplicity arises from the fact that by encapsulating the complex interacting state machines within a single component and expressing the inputs and outputs with temporally abstract signals, the result can be expressed with far fewer states. Yet it retains a useful degree of predictive power: for example, it predicts that pressing the keys `Local` and `B` will cause the brightness output to increase – a sufficient prediction that if the brightness does *not* increase, the right suspects will be identified.

5.9 Related Work

There are numerous formalisms, languages and programs for reasoning about time and change. For the present purpose it is sufficient to briefly identify four salient expressiveness and tractability issues and to point out that TINT takes an extreme position, always favoring tractability over expressiveness.

5.9.1 Temporally Quantified Statements

Systems that reason about time can in part be characterized in terms of the kinds of facts that they allow to be temporally quantified. Some systems admit only statements about parameter values, where the parameters may be either continuous or discrete quantities [Simmons83] [Bobrow85] [Williams86] [Kohane87]. Treating propositions as boolean valued functions (often called *time tokens*) allows any atomic proposition to be quantified [Dean87] [Shoham87]. There have been proposals to allow arbitrary first order sentences to be temporally quantified [McDermott82] [Moszkowski82] [Allen84], but there is no successful implementation of such a language. TINT "signals" fall into the first of these categories.

5.9.2 Intervals and Constraints on Intervals

Timestamping facts so that they hold at single time points is the most primitive form of temporal quantification, but this is hardly ever used except in a theoretical setting [Hanks86] [Shoham86]. A slightly improved scheme is that used in TINT and TCS [Russ86], in which statements hold over intervals with fixed numeric upper and lower bounds. Discovering intersections between intervals is trivial, but the expressiveness of such schemes is quite limited. One fundamental difficulty is that systems with feedback can result in runaway inference loops for which each new deduction only marginally extends the previous history. The alternative is to allow algebraic constraints of varying sophistication among the intervals on sparse and dense sets of points. The straightforward approach is to do so with inequalities on the endpoints of the intervals [Valdes86] [Williams86] [Kohane87] [Ladkin87]; a different approach is to use an algebra of intervals [Allen83] [Vilain86] [Valdes87]. With either approach, there is a tradeoff between expressiveness and the tractability of detecting interval overlaps; the more complex the constraints and the more complete the constraint propagator, the weaker the performance guarantees that can be made. Allen's constraint propagation scheme is a typical compromise: the propagator is $O(n^3)$ in the number of intervals but will not detect all inconsistent orderings, since the latter is NP-complete [Vilain86]. As noted earlier, TINT takes an extreme position in favor of tractability, thereby avoiding most of these issues. With fixed numeric bounds, detecting overlap is trivial, and while runaway inferences cannot be prevented they are at least easy to detect using bounds on the number of predications in each history.

5.9.3 Persistence

The world has inertia. Many programs for maintaining temporal assertions reflect this by building in implicit persistence of facts over time. For example, TMM [Dean87] will autonomously assume the persistence of any fact in order to answer queries. TINT and TCP [Williams86], on the other hand, do not; only the application program can add underived facts about the duration of intervals. The simple machinery in TINT never introduces new assumptions on its own, and so as a consequence there is an explicit justification for every prediction. This is just what is needed for troubleshooting.

5.9.4 Temporal Indexing

Database organization obviously has an impact on the kinds of queries that will be answered efficiently. A recurrent concern in temporal reasoning programs is how a database of temporally quantified statements should be indexed. A common approach is to organize all the intervals referring to a single parameter, token, proposition, or signal into a totally ordered list [Williams86] [Dean87]. An alternative is to organize the intervals into a hierarchy such that all the intervals at the leaves occur close together in time, irrespective of the propositions they refer to [Kahn77] [Dean87]. These schemes are not incompatible; in fact most systems use a multiple indices or a hybrid approach. TINT does not — it simply orders all the intervals referring to a given signal by increasing lower bounds.

5.10 Summary of Behavior Representation

TINT is a temporal reasoning system that propagates assertions about time-varying values at multiple levels of abstraction. The framework of *signals*, *abstractions*, and *behaviors* means that it can be very simple in its syntax, semantics, and computational machinery. There are three key reasons that TINT can be so simple and still allow the representation and troubleshooting of complex circuits.

First, there is a rich vocabulary of temporal abstractions with which to describe behavior. These temporal abstractions include such familiar concepts as *change*, *cycle*, and *frequency*. Good abstractions for troubleshooting preserve fidelity, strength, and efficiency by sacrificing precision. Temporal abstractions are good for representing digital circuits for troubleshooting because they can make the prediction task much more efficient, while preserving fidelity and precision for those signal properties that the troubleshooter can easily observe and that will be disrupted by typical failures.

Second, there are principles by which temporally abstract behavior definitions can be built for many circuits. Temporal abstractions result in strong abstract behaviors when the underlying behaviors are event preserving. Since not all components have behaviors that are not event preserving, the techniques of *reduction* and *synchronization* are ways of taking subsets of behavior that are event preserving. Encapsulating loops allows these former abstraction techniques to be applied to groups of connected components.

Third, there is an important distinction between the definitions of the behavior of individual components and the deductions that will be made about them during troubleshooting. There are many logical consequences of each abstraction and behavior definition that would lead to useless deductions during the prediction subtask of troubleshooting. TINT rules for each abstraction and behavior are included only when they make deductions about observable signals or when the deductions about signal values that they make hold over significant stretches of time.

Chapter 6

Representing Faults and Misbehaviors

The goal of a troubleshooting program is not mere generation of candidates, but efficient discrimination among them. However, there are three fundamental obstacles to efficient discrimination. First, the observations that the troubleshooter makes of the device may be imprecise. As a consequence it may be impossible to distinguish between some candidates. Second, some component behaviors may be so complex as to be intractable to reason about in any way other than from causes to effects. As a consequence the troubleshooting engine might not find all the conflicts derivable from the observations it has made and hence inconsistent candidates may survive. Third, even if reasoning from effects to causes is possible, there may be reasoning impasses that leave ambiguities resolvable only through intractable techniques. Again, the troubleshooting engine may not find all the derivable conflicts, so that inconsistent candidates may survive.

In the face of these fundamental difficulties a partial solution is to draw a distinction between the *possibility* of a candidate and its *plausibility* relative to other candidates. Instead of asking for the logically possible candidates, a more realistic goal is to ask for the most likely candidates among those possible. The program can then terminate when any desired degree of certainty is achieved, that is, after some diagnosis is significantly more likely than the others. As an additional benefit, the choices about which observations to perform will be more efficient because they will be biased toward discriminating between the most likely candidates, no matter what certainty is set as the termination goal. There is always the danger that estimates of relative likelihood will be inaccurate. It is possible that with bad estimates and a low threshold of certainty for termination, the program could terminate with an incorrect diagnosis. Commitment to using estimates of the likelihood of candidates implies a commitment to being circumspect about any decisions the program makes that are overly sensitive to those estimates. Nevertheless, even giving candidates crude likelihood estimates can provide a useful degree of bias.

Ranking candidates by their likelihood opens up new sources of knowledge to take advantage of. An obvious source of knowledge concerns the relative failure rates of the individual components in the candidates. These are ultimately grounded in accumulated statistical data but can also be partially derived from knowledge about the physical construction of the components. Another source of knowledge is *fault models* — knowledge not just about how often components fail, but also about how they usually fail and their misbehavior when they do. This kind of knowledge is used in a number of model-based troubleshooting programs including SOPHIE [Brown82] and IDS [Pan84].

In typical uses of fault models, each component has a set of misbehaviors that is assumed to be exhaustive; candidates can be ruled out by showing that none of their known misbehaviors are consistent with observations. But the crucial point is that the program does not need to have an exhaustive set of all the ways any given component can fail — it need not know any at all, in fact. However, if knowledge is available about a component misbehavior that can result from some physical failures and the proportion of failures in that component that would result in that misbehavior, then the troubleshooting engine can take advantage of it. By knowing one or two of the most likely failure modes of a component the program can make a better estimate of the likelihood that it is actually faulty. For example, suppose that telephone jacks fail in dozens of different ways, but that when they fail, half of the time the effect (the misbehavior) is as if all of the contacts were open circuits, and the other half of the time the effects are different. This knowledge can be used to adjust the likelihoods of candidates that hypothesize the jack is broken. If the observations of the circuit indicate that it would be inconsistent for all the contacts to be open then the jack is a relatively less likely, though still a possible candidate. No coverage has been sacrificed. The program has simply done what a human troubleshooter would do — it has brought to bear knowledge about the way things usually break to focus on the most likely possibilities.

Fault models, then, can be used as heuristics within a larger framework of failure likelihoods. Although this chapter is mainly about fault models, the first portion is spent presenting failure likelihoods as a partial solution to difficulties in discriminating candidates. Next, *syndromes* are presented as a refinement of that solution. Syndromes are the concrete representation in BASIL and TINT for the abstract notion of a fault model. They are added manually to the knowledge about a particular circuit; they are not learned or otherwise automatically generated. Next, several principles for the appropriate use of syndromes in representing circuits will be presented, along with examples appearing in the Console Controller Board. Finally, the consequences of using knowledge about syndromes in troubleshooting will be discussed. The mechanics of ranking candidates by likelihood and for using syndromes to adjust those rankings will be treated in the next chapter along with other details of the troubleshooting engine.

6.1 Failure Likelihoods

Estimating failure probabilities in general is subtle and complex; a very simple framework is used here. For example, independence between failures is assumed, a strong simplifying assumption (although not as strong as assuming that failure effects are independent). This simple framework is adequate because (as discussed later) the probabilities are used in such a way that the overall performance of the troubleshooting engine is relatively insensitive to small variations in these estimates.

The *status* of each BASIL component indicates whether it is believed to be physically damaged. The `status-of` predicate denotes this: when `[status-of U25 working]` is true means that chip U25 is believed to be undamaged. The status `other` means that the component is believed to be damaged in some way. A prior probability is assigned to the `working` status for each component, and the probability of having status `other` is then the difference between 1 and the probability of it working. As discussed in Chapter 2, these prior probabilities influence the ranking of candidates and probe suggestions produced by the troubleshooting engine: candidates involving the components with higher probabilities of having status `other` will appear to be likelier candidates, and the probes that the troubleshooter suggests will tend to be those that discriminate among the likelier candidates.

The probability of a given component working is estimated from its "complexity" — a nonnegative integer representing the number of breakable physical parts and how likely they are to break. Assuming independence, the probability of a component having status **working** is the probability that all its components are working. The probability of failure in a component with complexity 1 has been assigned .0001 — any number very close to 0 could have been used. Some typical probabilities for various components are shown below:

Component	Complexity	Probability of **working**
Etch	1	$.9999^1 = .9999$
Chiplet	1	$.9999^1 = .9999$
Pin	2	$.9999^2 = .9998$
16-pin Chip	33	$.9999^{33} = .997$
Oscillator Chiplet	100	$.9999^{100} = .99$
Console Controller[1]	≈ 2000	$.9999^{2000} = .82$

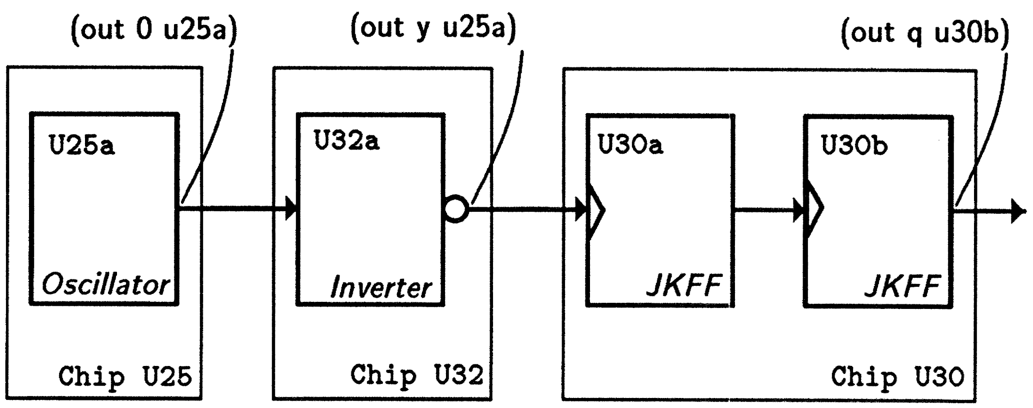
There are better ways of estimating failure rates; the power dissipation of the chip, for example, would probably be a better predictor. This scheme has the advantage that it can be derived from the representation of physical structure once a basic unit of complexity has been chosen.

The prior probabilities assigned to each component status influence the candidate rankings and probe suggestions. The likelihood of a candidate is the normalized probability that all the components in the device have the status assigned by that candidate. The Clock Generator provides a simple example. Assume that etches and chiplets other than the oscillator have complexity 0 so that their probability of working is 1.0; the three components and their likelihoods are then:

Component	Kind	Complexity	Probability of **working**
U25	Oscillator	100	$p(\text{U25}) = .9900$
U32	14 pin chip	28	$p(\text{U32}) = .9972$
U30	16 pin chip	32	$p(\text{U30}) = .9968$

[1] .0001 is actually too large, as can be seen from this anomaly. It is used only to simplify presentation.

Figure 6.1: Clock Generator



Suppose that a discrepancy is observed at (out q u30b), resulting in the conflict ⟨U25, U30, U32⟩. The candidates are the minimal covering sets [U25], [U30], and [U32]. The probability of each of these candidates is the probability that the named component is not working and that the others are. A weight for each candidate is then computed as the probability normalized with respect to all candidates:

Diagnosis		Likelihood	Weight
[U25]	$(1 - p(\text{U25})) \times p(\text{U30}) \times p(\text{U32}) =$	.00989	.63
[U30]	$p(\text{U25}) \times (1 - p(\text{U30})) \times p(\text{U32}) =$	.00315	.20
[U32]	$p(\text{U25}) \times p(\text{U30}) \times (1 - p(\text{U32})) =$	.00278	.17

As this example shows, candidates involving components with relatively higher failure likelihoods tend to end up with the largest weights. In this case the rankings are stable under perturbations in the component failure likelihoods so long as their ordering is maintained, that is, so long as the physical complexity of U25 is greater than that of U30, and of U30 greater than U32.

The troubleshooting engine can stop when there is one candidate above some threshold, which is usually almost 1. The relative proportions of the failure likelihoods among components can influence the decision to terminate. In the example above, if the threshold were set to .90, the program would terminate when one of the candidates had weight above .90. Had the physical complexity of U25 been 600 it would have had 90% of the weight and the program would have stopped, concluding that U25 was most likely to be broken. Note that it took more than an order of magnitude difference between complexity estimates — 600 being nineteen times as large as the complexity of U32 — to get this effect, however. Higher thresholds require bigger relative differences — for example, a threshold of .95 would have required the complexity of U25 to be 1100 for termination without further observations.

If no candidate is above threshold these candidate weights are used to help decide where the next probe should be made. To a crude first approximation, the choice of probe location will be biased toward places close to the higher ranked candidates. For example, in the example above `(out 0 u25a)` would be chosen over `(out y u32a)` as long as the complexity of U25 was greater than that of U30. The details of probe selection will be presented later; the important point for the moment is that the better the estimate of component failure likelihoods the fewer probes will be needed on average in the long run.

Using failure likelihoods provides an incremental improvement in the ability of a troubleshooting engine to distinguish candidates. By presenting the plausible candidates in addition to the possible ones and biasing the observations made in favor of the likely failures, the troubleshooting engine should be able to provide the right diagnoses most of the time using fewer probes.

6.2 Representing Syndromes

Fault models provide an additional increment of power to the troubleshooting engine because they can be used to make better estimates of candidate likelihoods. Roughly, this is done by (i) splitting the weight assigned to a given candidate into portions, one for each way that some component in that candidate might be misbehaving, and (ii) showing that one or more of those portions corresponds to an inconsistent diagnosis. If (ii) succeeds this means that that component was not as likely to be broken as was thought. That candidate will be made relatively less likely, thereby indirectly boosting the other candidates. The details of how the troubleshooting engine performs these steps will be presented in the next chapter. The present concern is the representation of how components misbehave and how likely they are to do so.

A *syndrome* is a set of sets of physical failures that result in equivalent misbehaviors of a component. Since the misbehavior of a component is relative to its intended behavior, each syndrome is thus tied implicitly to a level of behavioral abstraction. For example, consider an imaginary chip **`inverter-chip`** with four pins (power, ground, input, output) and just one inverter on it. Some of the following are physical failures inside the chip: (a) the pulldown is open (b) the output pin is open (c) the pullup is shorted (d) the pulldown is shorted (e) the input pin is open. Three example syndromes are:

1. Several different combinations of physical failures would cause the inverter to produce a constant output logic-level of `1`. Its pulldown might be open, its output pin might be open (since TTL floats high), its pulldown might be open *and* its pullup shorted, and so on. This is the set of sets {{a}, {b}, {a,c} ...}.

2. Another set of combinations of failures cause the inverter to produce a constant logic-level of `0`. Its input pin might be open, its pulldown might be shorted, and so on. This is the set of sets {{e}, {d}, ...}.

3. Both sets of failures described above cause the inverter to produce a constant frequency of 0. The union of those sets is thus another syndrome. This is the set of sets {{a}, {b}, {a,c}, {e}, {d} ...}. Although in principle syndromes can thus intersect, in practice the syndromes for a given component are disjoint sets.

Syndromes are sets of sets of failures, but for mnemonic value they are usually named according to the misbehavior that results. For example, syndrome 3 above, which caused the inverter-chip output frequency to be zero, will be denoted **`zerof`**.

The **`status-of`** predicate is used to indicate the belief that a given component has a particular syndrome. Thus **`[status-of i zerof]`** says that component `i` has some physical failure among the set causing it to output a constant frequency of zero. The status **`working`** corresponds to an empty set of failures; the predication **`[status-of i working]`** says that the physical component `i` has no failures and is working perfectly.

An estimated likelihood is assigned to each of the possible statuses of a physical component, using the complexity estimates introduced earlier. For example, assume that pins have complexity 2 and everything else has complexity 0. Then the likelihood that the inverter-chip is `working` is estimated as $.9999^8$ — the likelihood that all four pins are working. The likelihood that the inverter-chip has syndrome `zerof` is estimated as $4 \times ((1 - .9999^2) \times .9999^6)$ — the likelihood that exactly one of the four pins is independently broken. This is only an estimate, since on the one hand there might be failures in the pins other than opens, but on the other hand multiple pin failures that would cause the same syndrome are not being counted. Finally, the likelihood that it has status `other` is then 1 minus the likelihoods of these other two statuses:

Inverter-Chip Syndrome	Likelihood	
`working`	$.9999^8 =$	.9992
`zerof`	$4 \times ((1 - .9999^2) \times .9999^6) =$	.0007
`other`	$1 - .9992 - .0007 =$	.0001

The troubleshooting engine can use this information to try to reduce the likelihood of candidates involving inverter-chip components. Suppose there were a candidate corresponding to a particular inverter-chip `i` being broken. This candidate and its weight would be split into two portions — one corresponding to the hypothesis that `i` had status `zerof`, the other to the hypotheses that `i` had status `other`. Suppose its weight had been .40. To a first approximation the weight would be split proportionately among these two according to their relative likelihoods .0007 and .0001, in this case .35 and .05. Now, if observations indicate that `i` cannot have status `zerof` the weight of that portion (.35) would get redistributed among all candidates. For example, suppose there had been two other candidates each with weight .30; after redistributing the weight .35 evenly across the three candidates, two would have weights of .42 each and the candidate involving `i` would have weight only .17. Thus the likelihood of `i` being broken relative to the other candidates will have been decreased from .40 to .17. The details of how this is done are presented in the next chapter.

To gain anything from a syndrome the behavior model must be able to detect that it is inconsistent with observations that the troubleshooter has made. Thus each component status has consequences in the behavior model. Recall that if a physical component has the status `working`, has power, and so on, then its mode is `normal`. In the case of the `inverter-chip`, for example:

```
  If  [isa ?x inverter-chip]
 and  [status-of ?x working]
 and  [thru ?l ?u (power (in power ?x)) t]
Then  [thru ?l ?u (mode ?x) normal]
```

Having a status of `zerof`, however, implies a mode of `inactive` no matter whether the component has power or not:

```
  If  [isa ?x inverter-chip]
 and  [status-of ?x zerof]
Then  [thru -∞ +∞ (mode ?x) inactive]
```

In the `inactive` mode the output frequency is zero:

```
  If  [isa ?x inverter-chip]
 and  [thru ?l ?u (mode ?x) inactive]
 and  Signal (fww ?w ?c (ll (out y ?x))) exists
Then  [thru ?l ?u (fww ?w ?c (ll (out y ?x))) 0]
```

The indirection from the status of "zerof" to the mode of "inactive" makes writing behavior rules more convenient. For one thing, the status of a component has no temporal bounds, but the `mode` signal does. For another thing, only physical components are given failure syndromes, while only functional components have behaviors. Finally, there are other ways of being in `inactive` mode, such as losing power:

```
  If  [isa ?x inverter-chip]
 and  [status-of ?x ?anything]
 and  [thru ?l ?u (power ?x) nil]
Then  [thru ?l ?u (mode ?x) inactive]
```

The following section will clarify this by giving examples of several syndromes and their associated misbehaviors.

6.3 Principles for Using Syndromes

There are two situations in which it is advantageous to represent syndromes and misbehaviors explicitly: (i) when there are functional components that have faults with unusually high likelihoods, or (ii) when the resulting misbehavior is drastically simpler than the correct behavior.

Faults with high likelihood are worth including explicitly. It is useful to know about very likely failures because if a particular component is one of many suspected of failure, but (say) 99% of the failures in components of that type produce a behavior other than the one being observed, then that component is almost certainly not the culprit.

One of the most common failures in the field occurring in digital circuits is the disconnection of a bonding wire. In BASIL, bonding wires are considered part of pins. The effect of breaks in them is to make the pin act as an open circuit. Thus one of the syndromes for pins is termed `open`, and its behavioral impact is to make the currents into both ends of the pin be `0` (the signal `(qci ?port)` denotes the sign of the current into `?port` and is discussed in Appendix E):

```
  If  [conn ?pin (hole ?i ?e) ?port]
 and  [status-of ?pin open]
Then  [thru -∞ +∞ (qci (hole ?i ?e)) 0]
 and  [thru -∞ +∞ (qci ?port) 0]
```

For example, if the externally visible node of this pin is connected to a pullup and should be pulled down via this pin, and the node is at logic level `0`, then the pin is probably not faulty. This is because if the pin were open, the node would be pulled up to `1`.

The likelihood of a pin working was earlier set to $.9999^2 = .9998$; the likelihood of it having status `open` is set to .0002. This makes the `other` status have likelihood 0:

Pin Status	Likelihood
`working`	0.9998
`open`	0.0002
`other`	0.0

Thus the pin is an extreme example of a component with a "likely" syndrome — it accounts for 100% of the failures in pins. It is exceptional in that respect, however; no other component has such a syndrome. The point stands, however, that it is useful to know about just because it is so likely.

Faults that drastically simplify behavior are worth including explicitly. One kind of "drastic simplification" of behavior is when the faulty component produces a constant output for all time, instead of responding to changes on its inputs.

For example, a common failure is that crystal oscillators crack or become loose in their casings; the result is that the output does not oscillate, but instead stays constant:

```
  If  [isa ?o oscillator]
 and  [thru ?l ?u (mode ?o) inactive]
 and  Signal (fww ?w ?c (ll (out 0 ?o))) exists
Then  [thru ?l ?u (fww ?w ?c (ll (out 0 ?o))) 0]
```

Thus for example, if the output of the oscillator is active it is probably not faulty. The syndromes and their likelihoods are based on the presumptions that oscillators fail about 50 times as often as pins, and that there is a nonzero likelihood that the oscillator may fail in other ways:

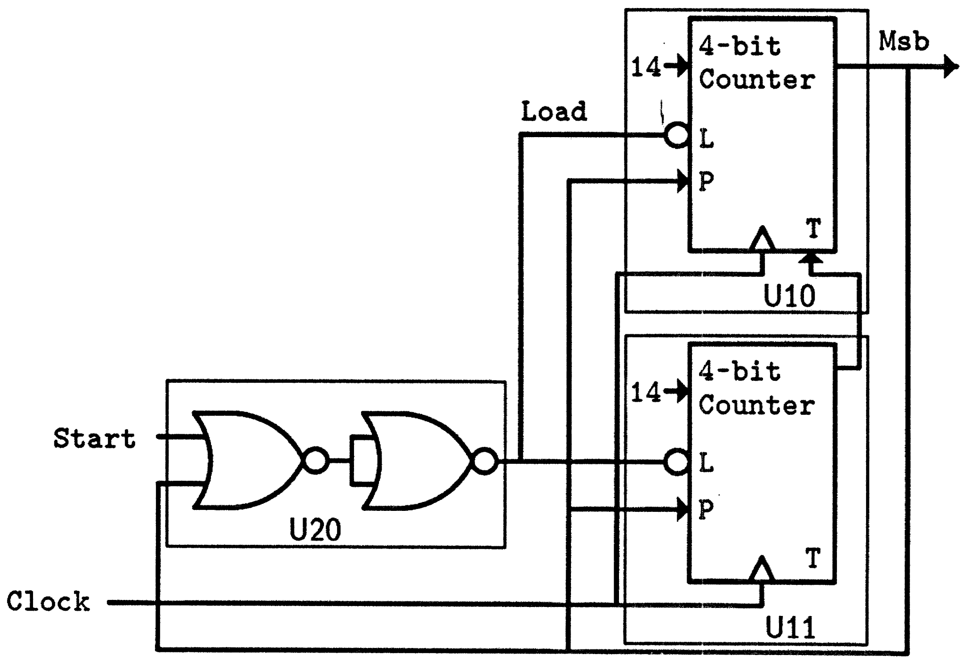
Oscillator Status	Likelihood	Description
working	0.99	$= .9999^{100}$
open	0.0099	$= 100 \times ((1 - .9999) \times .9999^{99})$
other	0.0001	

The syndrome is useful because the misbehavior that results is simple and sufficiently different from what is expected that it does not require much additional reasoning to detect whether it is consistent with observations or not. Had the syndrome been that the oscillator (say) skipped every hundredth cycle, a detailed model of behavior would have been required to represent it, and the available observations would not have been able to distinguish it anyway. Such misbehaviors are usually better dealt with at the lower levels of physical and behavioral detail from which they originated.

Useful syndromes have both of these properties — common and simplifying. In the case of the pin and oscillator these properties are achieved because of the physical simplicity of the components. These properties can also be achieved in functional components with more internal structure and complex behavior. Syndromes can have high likelihood if many internal faults produce the same overall misbehavior. Faults can cause the behavior to be drastically simplified if they dominate all the outputs of the component, or if they lie on internal sequential feedback paths so that the effects of local misbehaviors aggregate and cascade. Thus, if there are several faults that cause the same misbehavior, and the misbehavior is simpler than the normal behavior — by having fewer reachable states, for example — then those faults constitute a useful syndrome.

Consider for example the burst detector in the Audio Decoder (Figure 6.2). Eighteen clock cycles after the **start** signal falls, the output **Msb** is asserted for one cycle.

Figure 6.2: Audio Counter



The internal structure of the burst detector involves three chips — two four-bit counters U10 and U11, and a quad NOR gate chip U20. Any of the three chips U10, U11, or U20 could fail in ways that prevent the burst detector from ever starting to count, so that **Msb** would always be 0. For example, there are three pins in U20 that if open would cause the Load signal to be stuck at 1, the result being that counting would never start[2]. Thus each of the three chips has a syndrome denoted **csb-inactive**, and if any of them have that status then the burst detector is **inactive**:

```
  If  [status-of ?u csb-inactive]
 and  (member ?u '(u10 u11 u20))
Then  [status-of csb01 inactive]
```

If the burst detector is in **inactive** mode then both its outputs are 0:

```
  If  [isa ?csb clocked-serial-burst-detector]
 and  [thru ?l ?u (mode ?csb) inactive]
Then  [thru ?l ?u (ll (out wr ?csb)) 0]
 and  [thru ?l ?u (ll (out clk ?csb)) 0]
```

For each of the three chips, the likelihood of each syndrome occurring is estimated from the likelihood of failures in the pins. For example, the likelihood of U10 working is $.9999^{32}$, the likelihood that all 16 pins are working. The likelihood of U10 having syndrome **csb-inactive** is $3 \times (.0002 \times .9999^{30})$, the likelihood that the chip has exactly one of the three single-pin faults that cause **csb-inactive**. The likelihood of **other** is just the residual:

U10 Status	Likelihood	Description
working	0.997	All 16 pins working
csb-inactive	0.0006	Any of 3 pins open
other	0.0024	

For U11, there are 4 open pin faults that can cause the syndrome:

U11 Status	Likelihood	Description
working	0.997	All 16 pins working
csb-inactive	0.0008	Any of 4 pins open
other	0.0022	

For U20, 5 open pin faults can cause it:

[2]This was checked by SSIM, a simple event-driven digital simulator that uses BASIL as its structure description language.

U20 Status	Likelihood	Description
working	0.997	All 14 pins working
csb-inactive	0.001	Any of 5 open pins
other	0.002	

The impact of this syndrome is that if it can be shown that it is inconsistent for the clocked serial burst detector to be inactive, then the likelihoods of candidates involving U10, U11, and U20 will be reduced somewhat — each by about one-fourth. The likelihoods of syndrome `csb-inactive` appearing in each of the three chips do not differ by enough to have any significant impact on the likelihoods of candidates containing U10, U11, and U20 relative to one another.

Another example in the Audio Decoder is the Manchester-to-serial decoder; it is a sequential circuit entirely encapsulated within the chip U12. When the chip U12 has status `mts-inactive` then MTS01 has status `inactive` as a consequence:

```
  If  [status-of u12 mts-inactive]
Then  [status-of mts01 inactive]
```

In the `inactive` mode, the serially encoded output of MTS01 has zero amplitude:

```
  If  [isa ?mts manchester-to-serial]
 and  [thru ?l ?u (mode ?mts) inactive]
 and  Signal (max-min-ww ?w (cs (out y ?mts))) exists
Then  [thru ?l ?u (max-min-ww ?w (cs (out y ?mts))) 0]
```

The likelihood of each syndrome for U12 is based on the fact that U12 has 20 pins, faults in 9 of which can cause the syndrome `mts-inactive`:

U12 Status	Likelihood	Description
working	.996	All 20 pins working
mts-inactive	.0018	Any of 9 pins open
other	.0022	

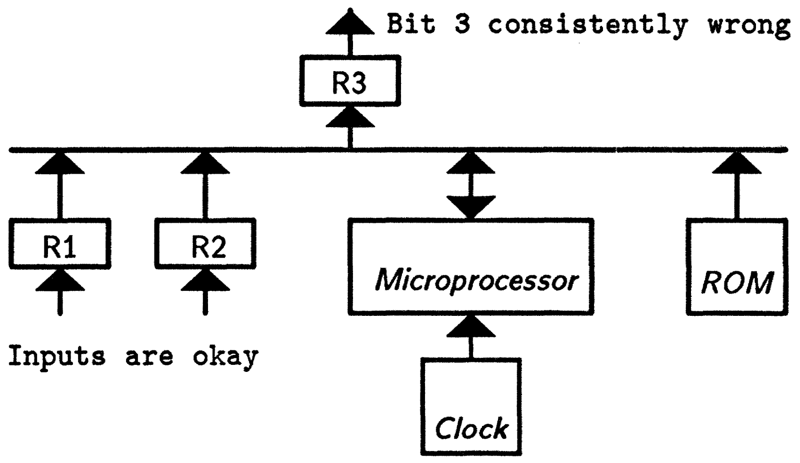
6.4 Consequences of Using Syndromes

By helping to discount unlikely misbehaviors, syndromes help a troubleshooting engine to ask for fewer observations, and this in turn makes troubleshooting complex digital circuits more feasible. For example, in the Audio Decoder one of the cases requires 9 observations without syndromes to arrive at a single-fault diagnosis, but 2 observations to arrive at the same diagnosis if the syndromes are included. Since the cost of making observations is generally assumed to be greater than that of extra computation, even more modest gains are worthwhile. The reduced number of observations is possible because the syndromes reduce the relative likelihoods of faults in the Manchester-to-serial converter and in the burst detector, and the troubleshooting engine is generally biased away from suggesting observations in the vicinity of components that are judged unlikely to be causing the observed symptoms.

Knowledge about how components misbehave is essential in troubleshooting complex circuits because the number of logically possible (but unlikely) misbehaviors and the amount of detail in the observations needed to track them down are so large. The effectiveness of fault models in providing focus stems from two sources, one general and one specific to digital systems. First, sometimes it is much easier to reason forward from causes to their effects than the reverse. The consequence is that it is easier to consider the ways a component might plausibly misbehave and rule them out individually, than to try and logically rule out all of them at once. Second, some behaviorally complex digital components have many internal faults that all result in the same few temporally abstract misbehaviors. The beneficial consequence is that if these few misbehaviors can be ruled out, the complex component will be judged an unlikely candidate.

As an example of the problems that result from the inability to reason from effects to causes, consider Figure 6.3. It shows a microprocessor dedicated to running a program that multiplies the contents of two external registers R1 and R2 and writes the result to R3. If the troubleshooter observes that the output register R3 has bit 3 consistently wrong, it will suggest not only that this register might be broken, but that the microprocessor, the read-only memory where its instructions are stored, the clock generator that runs the processor, and so on, could all be broken. Intuitively, these other candidates are implausible; it might be logically possible for the microprocessor to be doing arithmetic incorrectly, or for the clock to be skipping

Figure 6.3: Every Component is a Candidate



cycles, or for some instruction to be slightly wrong, but if these things were happening the observed misbehavior would probably be much more drastic than just the one wrong bit. For example, if the microprocessor is adding numbers wrong it is likely to make a wild branch to a location containing an illegal instruction. If it could be inferred from observations of the outputs of the microprocessor that its instructions from the ROM were correct, or that the clock output was correct, those candidates would not get proposed. But logically speaking such inferences are unfounded, because it *could* in principle happen that way — it is just very unlikely.

The microprocessor example also illustrates why knowledge about syndromes is useful in complex digital circuits. A discrepancy at the output of R3 in principle implicates the microprocessor, ROM, and clock generator, and requires observations to determine whether the clock is running or not, whether all the ROM locations have the right value, and so on. But experienced human troubleshooters would examine the inputs and outputs of the registers first — and probably find the problem there very quickly. Experienced troubleshooters, upon seeing a digital circuit perform some function correctly, tend to exonerate (at least temporarily) the complex portions of the circuit. The usual expectation is that any failure there will result in a catastrophic rather than a subtle misbehavior. Sequential circuits tend to have "inactive" syndromes associated with them and because the circuit did *something*, that syndrome was ruled out. In the present example, the microprocessor gets exonerated because the output of register R3 is at least changing. In other domains this heuristic might not work, for example in analog domains in which failures usually have more subtle effects. The context, however, is troubleshooting digital circuit boards, and many of the failures there are not at all subtle.

The reason that digital circuits misbehave this way stems from aspects of their design. Complex functions tend to get implemented in state machines or as firmware for general processors. The circuits then use the same hardware components over and over to implement different steps of the overall computation, many of which depend on the previous step. Hence a perturbation caused by failure in any one unit of hardware rapidly cascades and propagates its effects. The very economy of the design — the reuse of hardware for different substeps of a complex behavior — means that after many cycles the behavior will little resemble that intended. Since complex components communicate with one another through protocols and languages in which the meaningful message sequences occupy only a fraction of the theoretically available bandwidth, when a component is intended to produce a message sequence understandable by some other component, the message will probably never get through. To extend the example, suppose the microprocessor must initialize some slave hardware by setting up sixteen eight-bit registers one at a time. If the master processor makes even one wild branch, or one bit is stuck on the data bus, the likelihood that the slave got a correct initialization message is rather slim.

Fault models are thus a powerful form of heuristic in troubleshooting complex digital circuits, both because of the general property that they tend to focus the model-based troubleshooting program on likely failures, and because of the specific property that the design of the digital circuits means that they can be treated as unlikely suspects if they perform even a portion of their intended behavior. As the behavioral complexity of field replaceable components increases, the more valuable this latter phenomenon becomes, since the model-based troubleshooting program can thereby avoid having to reason in detail about their internal structure and behavior.

6.5 Summary of Faults and Misbehaviors

Experience with model-based troubleshooting has shown that with increasing behavioral complexity, approaches that avoid the use of fault models have little utility in the real world because the problem of isolating a component in the face of limited observability and behavioral complexity is often inherently underconstrained [Hamscher84]. Ideally there is unlimited observability, every component has behavior that is easy to manipulate algebraically, and computation is so cheap that competing diagnoses can be discriminated through computationally intensive techniques such as exhaustive case splitting over finite fields of values. For devices of any interesting complexity these are not realistic approaches. A partial solution is to limit consideration of diagnoses to those that are plausible, rather than considering all that are logically possible. With this more limited goal, fault models can be seen as heuristics for refining estimates of component failure likelihoods. In BASIL and TINT, fault models are called syndromes, and have both physical and functional aspects. The syndromes to be included for each component type are chosen on the grounds of likelihood and simplicity: they should account for a significant fraction of failures in components of that type, and they should result in drastically simplified behaviors. While total reliance on fault models for automated diagnosis has serious drawbacks, it does not follow that they have no role in model-based troubleshooting. In the case of digital circuits in particular, fault models turn out to be powerful heuristics because the very design of complex digital systems means that fault effects result in misbehaviors that are catastrophic, easy to detect, and easy to rule out.

Chapter 7

Troubleshooting

The representations of structure and behavior discussed in earlier chapters are heavily influenced by their intended use in model-based troubleshooting, in particular, by their intended use with the troubleshooting engine XDE[1]. Like GDE [deKleer87], XDE works by (i) tagging each prediction made by the behavior model with its set of supporting assumptions, (ii) recording conflicts among the consequences of these assumptions, (iii) constructing the set of candidates (some possibly indicating multiple faults) as the minimal covering sets of those conflicts, and (iv) suggesting as the next observation the one expected to most reduce the uncertainty among the set of candidates.

XDE extends this procedure by adding two new operations that can be performed before suggesting new observations: *decomposition*, which enables hierarchic diagnosis, and *refinement*, which enables the use of fault models. Decomposition and refinement are integrated into the procedure with decomposition having priority over refinement, which in turn has priority over probe selection. XDE constructs candidates that are assigned weights according to their relative likelihood. Those with weight above 10% are eligible for refinement and decomposition. After each new observation, XDE finds the most likely candidate and *refines* it. Refinement involves selecting the most likely syndrome for a component believed faulty in that candidate, and predicting the effects of that syndrome. If there is no such refinement operation available, it *decomposes* a component instead. If no diagnosis is eligible for this either, it suggests a probe.

[1]eXtended Diagnostic Engine.

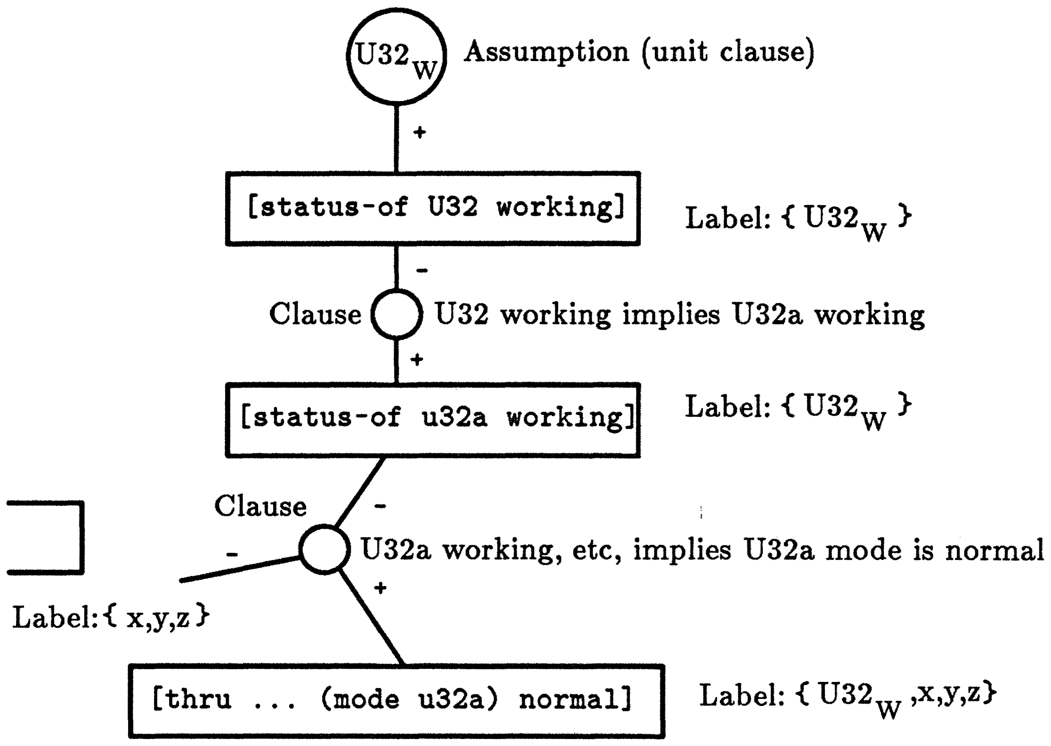
This chapter presents XDE and its interaction with the representation choices made in the structure language BASIL and behavior language TINT.

7.1 Conflicts and Candidates

XDE inherits the terminology of *assumptions*, *environments*, *conflicts*, and *candidates* from GDE, interacting with BASIL and TINT mainly through **status-of** predications.

In TINT an assumption is a unit clause supporting one predication. For example, let $U32_W$ denote the assumption that chip U32 has the status "Working." $U32_W$ is a unit clause attached to the single predication [status-of U32 working] (top of Figure 7.1).

Figure 7.1: Predications, Assumptions, and Environments



Environments are sets of assumptions; for example, $\{\text{U32}_\text{W}\}$ is the environment in which chip U32 is assumed to be working. The predication `[status-of U32 working]` could be true in more than one environment, and the set of environments in which it is true is called its label. For example, there could be another assumption that the entire board is working; `[status-of U32 working]` would be true also in the singleton environment consisting of that assumption.

A clause is a disjunction of predications. When a clause is installed connecting two or more predications, some predications may become true in new environments. For example, inverter chiplet U32a is part of chip U32. The clause $\neg$`[status-of U32 working]` $\vee$ `[status-of U32a working]` would be installed (middle of Figure 7.1), since if U32 is working then all its subparts including U32a must be working. Because of this clause, `[status-of U32a working]` would become true in the environment $\{\text{U32}_\text{W}\}$.

TINT rules fire on predications and make deductions in the form of (usually new) predications. Each firing results in the installation of a clause connecting the old predications to the new predication. For example, suppose a rule fired to deduce that the mode of U25a was normal, and installed a clause to that effect. The new predication would then be true in the environment that was a union of the environments of the old predications (bottom of Figure 7.1). Ultimately, any consequence of assuming that U25 is working will have some superset of the environment $\{\text{U32}_\text{W}\}$ in its label.

If TINT makes two different deductions about the value of a signal at a certain time, a conflict is recorded. At least one of the assumptions underlying those deductions must be false. The conflict is the union of the environments of the contradictory deductions and is denoted $\langle \ldots \rangle$. For example, if a certain signal was supposed to be 10^6 in the environment $\{\text{U25}_\text{W}, \text{U32}_\text{W}\}$ but it was supposed to be 0 in the environment $\{\text{U30}_\text{W}\}$, then the union $\langle \text{U25}_\text{W}, \text{U32}_\text{W}, \text{U30}_\text{W} \rangle$ is a conflict.

All of the assumptions that XDE makes are about the statuses of physical components, hence all the candidates that it produces are sets of physical components corresponding to repairs. For example, if the above conflict were the only conflict known, then the candidates are its minimal covering sets $[\text{U25}_\text{W}]$, $[\text{U32}_\text{W}]$, and $[\text{U30}_\text{W}]$. At least one of the chips U25, U32, or U30 needs to be replaced.

7.2 Decomposition

In hierarchic diagnosis, a component suspected of being faulty can be decomposed to reveal its subcomponents. The decomposition of a component involves two conceptually separate operations: (i) firing the behavior rules for the subcomponents, which usually refer to signals at a different level of abstraction than that of their parent, and (ii) making the troubleshooting engine entertain fault hypotheses about each individual subcomponent, rather than about the parent. In traditional hierarchic diagnosis these two operations are usually considered identical. That works fine within a single strict hierarchy, as in HT [Davis84] and DART [Genesereth84]. To deal with the physical and functional hierarchies in BASIL, however, it is advantageous to draw a distinction between the two operations.

To make the TINT behavior rules for a certain component fire requires creating an explicit `status-of` predication for it. This operation is called *instantiation.* Instantiating inverter U32a, for example, creates the predication `[status-of U32a working]`. Rules about the mode and behavior of U32a will only fire after `[status-of U32a working]` becomes true. Since U32a is a part of chip U32, if U32 is believed to be working then U32a should be believed to be working too. Thus a clause linking the two is installed, as illustrated earlier in Figure 7.1. Also, the parent component should be believed to be working if all its subcomponents are. When all of the subcomponents of a parent component have been instantiated, another clause is installed that makes the parent `status-of` predication true if all the subcomponent `status-of` predications are.

After instantiating all of the subcomponents of a parent component, XDE will not construct candidates involving those subcomponents until an assumption (unit clause) has been created for each of them. After being created these new assumptions will then appear in the labels of some predictions about the behavior of the device, will appear in conflicts, and thus will appear in candidates. The parent component will have the status `working` in the environment consisting of all the assumptions about its subcomponents (unless that environment is itself a conflict). Any assumptions about the original parent component are no longer needed and can be deleted[2]. This

[2]The binary clauses of the form $\neg parent \vee child$ are deleted too, a detail that improves the efficiency of the TMS.

operation is called assumption *splitting* — any assumptions about the status of a component are deleted and one assumption is created for each of its instantiated subcomponents.

Suppose devices were represented using only one component hierarchy. If a top-level component were a candidate, then its subcomponents would be instantiated and some rules would run. Then the assumption that the component is working would be split and new conflicts would be discovered involving its subcomponents. Some of the subcomponents would then appear in candidates. Each of these could then be treated recursively — their subcomponents instantiated and their assumptions split.

It is helpful for all the assumptions present at any moment to be independent, since this simplifies candidate ranking. If the hierarchy is not guaranteed to be strict, it takes extra work to ensure that each pair of assumptions is independent, since any pair of assumptions might refer to two components that share subparts. If the hierarchy is strict, at each descending step it is easy to guarantee that this never happens. Thus it is useful to locate assumptions only within strict part-of hierarchies.

Now suppose that there are two hierarchies and that there is no obvious correspondence between nodes in the two. BASIL, for example, has physical components and functional components in separate hierarchies that meet at their leaves. Figures 7.2 through 7.4 show an example. There are two boards A and B, each having several chips. Three of the chips on A and two of the chips on B form a single four-bit adder. The four-bit adder is composed of two two-bit adders tb1 and tb2. Each two-bit adder is composed two full-adders, each full-adder is composed of two half-adders and an OR gate, and each half-adder is composed of an AND gate and an XOR gate. Each of the full-adders fa1 through fa4 is distributed across three chips — a quad AND gate chip, a quad XOR gate chip, and a quad OR gate chip.

In BASIL, assumptions about the status of components are attached to physical components. This suggests that the diagnosis proceed top-down through the physical hierarchy, always staying as high as possible. However, TINT behavior rules are attached only to the components in the functional hierarchy. While descending through the physical hierarchy, it makes sense to fire the behavior rules for ever more detailed functional components. Since there is no obvious correspondence between the components in the two hierarchies (Figure 7.5), there is a coordination problem — how deep into the functional hierarchy should components be instantiated for each newly split

Figure 7.2: Physical Organization of Four-Bit Adder

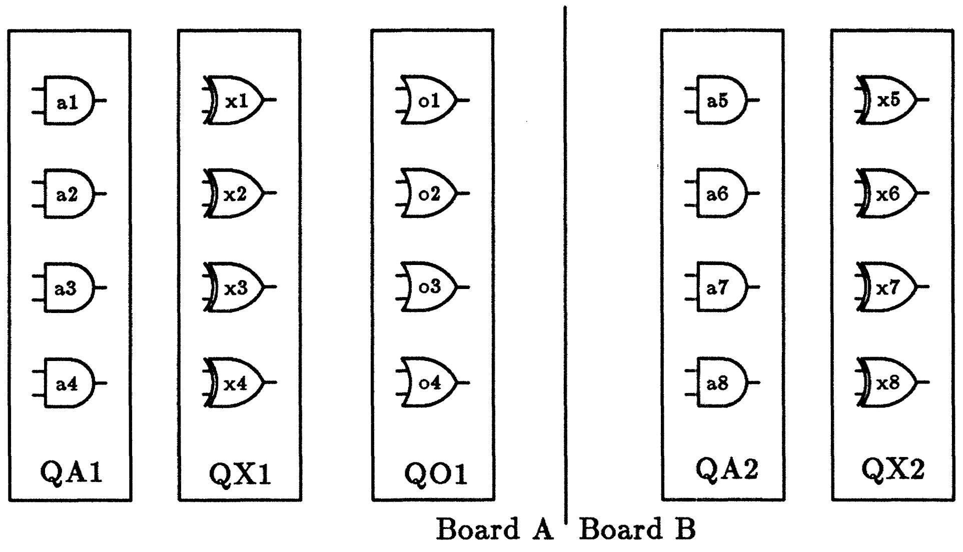


Figure 7.3: Functional Organization of Four-Bit Adder

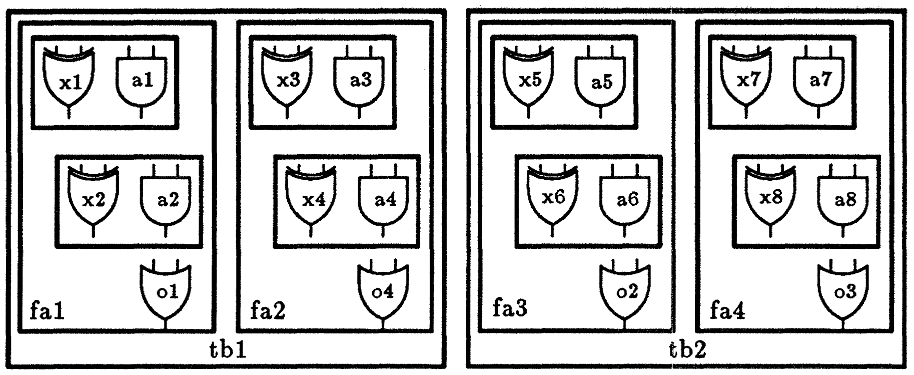
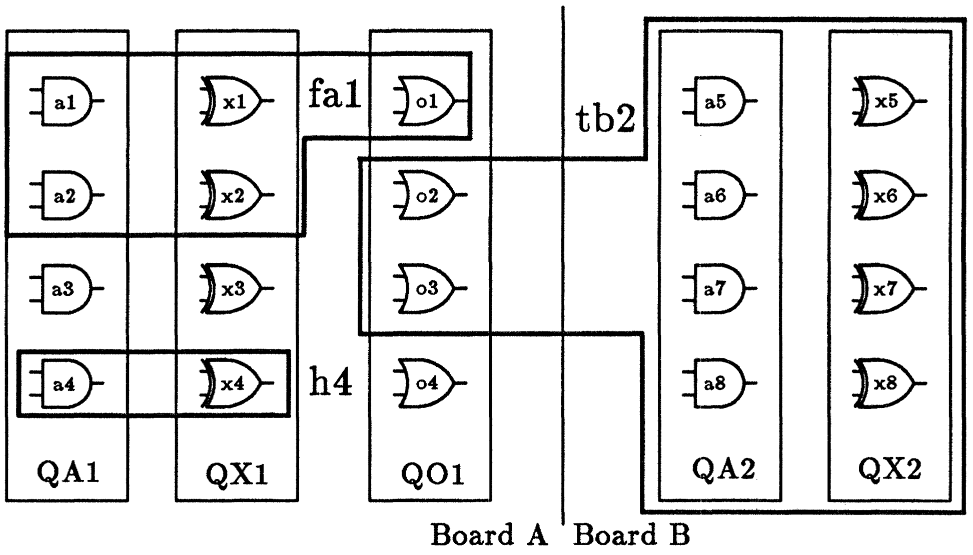


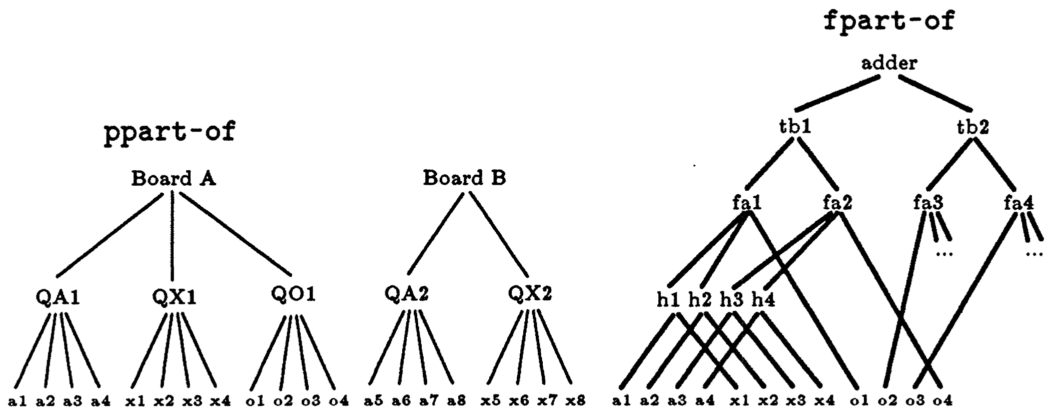
Figure 7.4: Physical and Functional Organizations



assumption in the physical hierarchy?

For each physical component, there is some functional component that fully contains it. For example, chips QA1 and QX1 are fully contained within the two-bit adder tb1. Chip QO1 is fully contained only within the whole four-bit adder. When chip QA1 has an assumption attached to it so that it can appear in diagnoses, rules should at least be getting run for every component that fully contains it. But this is not deep enough, since there would never be enough behavioral detail to distinguish between diagnoses involving that physical component and others contained by the same functional component. For example, if only the rules at the level of two-bit adders were being run, there would be no way to detect a conflict in which QA1 appeared but QX1 did not. This is because QA1 and QX1 must both be working for either of the two-bit adders to be working. Going one level deeper in the functional hierarchy would not help — at the level of full-adders there is still no way to find a conflict involving QA1 but not QX1, since both must be working for full-adders fa1 and fa2 to work. Going one level deeper in the physical hierarchy, however, would help: with QA1 is assumed to be working, rules would be run for any components that fully contain any of its

Figure 7.5: Physical and Functional Decompositions of the Four-Bit Adder

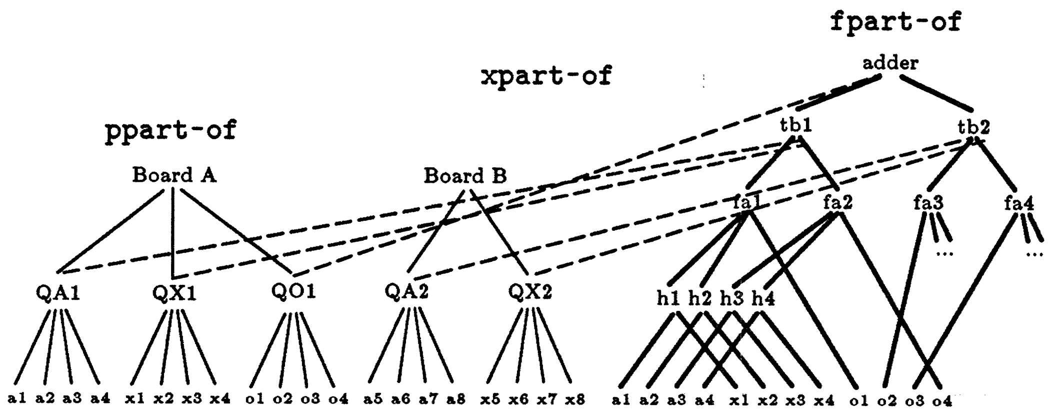

fpart-of
adder
tb1
tb2
fa1
fa2
fa3
fa4
h1 h2 h3 h4
ppart-of
Board A
Board B
QA1
QX1
QO1
QA2
QX2
a1 a2 a3 a4 x1 x2 x3 x4 o1 o2 o3 o4 a5 a6 a7 a8 x5 x6 x7 x8
a1 a2 a3 a4 x1 x2 x3 x4 o1 o2 o3 o4

subcomponent AND gates a1, a2, a3, or a4. In this case, the corresponding functional components happen to be the gates themselves, and the behavior rules at the level of gates have enough behavioral detail to detect conflicts involving QA1 without involving QX1.

This yields the criterion that XDE uses to decide how deep in the functional hierarchy to run rules, given a certain level of assumption in the physical hierarchy: instantiate all functional components that fully contain any immediate physical subcomponent. A physical component is "fully contained" if it is a *physically maximal part-of* the functional component, abbreviated **xpart-of**. The **xpart-of** relation holds between each physical component and zero or more functional components (Figure 7.6). A physical component is a physically maximal part of a functional component when it all its subcomponents help to implement that functional component. Strictly speaking, it is when all the leaf **ppart-of** descendants of the physical component are leaf **fpart-of** descendants of the functional, but the parent of the physical component is not maximal. For example, QA1 is **xpart-of** tb1 because all of its leaf subcomponents are leaf subcomponents of tb1, but the same is not true of the parent of QA1, Board A. Hence if Board A were assumed to be working, QA1 is an immediate physical subcomponent of Board A and is **xpart-of** tb1, so tb1 would be instantiated. The children of tb1 would not.

There is one further complication, which is that for each layer of physical

Figure 7.6: XPART-OF Relations in the Four-Bit Adder


fpart-of
adder
xpart-of
ppart-of
tb1
tb2
Board A
Board B
fa1
fa2
fa3
fa4
QA1
QX1
QO1
QA2
QX2
h1 h2 h3 h4
a1 a2 a3 a4 x1 x2 x3 x4 o1 o2 o3 o4 a5 a6 a7 a8 x5 x6 x7 x8 a1 a2 a3 a4 x1 x2 x3 x4 o1 o2 o3 o4

detail, there may be several layers of functional detail, and XDE proceeds through the functional detail one level at a time. The "decomposition" operation may thus be applied to the same physical component more than once, although sometimes it will result in functional components being instantiated, and other times in splitting of assumptions. The table below shows an example of the order in which XDE would intersperse assumption splittings and component instantiations.

Step	All Existing Assumptions	New Instantiations of Functional Components
1.	A, B	
2.		adder
3.		tb1, tb2
4.	QA1, QX1, QO1, B	
5.		fa1, fa2
6.		h1, h2, h3, h4, o1, o4
7.		a1, a2, a3, a4, x1, x2, x3, x4
8.	A1, A2, A3, A4, QX1, QO1, B	
9.	A1, A2, A3, A4, X1, X2, X3, X4, QO1, B	

Step 1: both boards are assumed working and no components are instantiated. Step 2: the adder is instantiated. Suppose the conflict ⟨A, B⟩ results.

Now [A] and [B] are candidates. Step 3: the subcomponents of the adder, tb1 and tb2, are instantiated. No further progress can be made in the functional hierarchy. Step 4: split the assumption that A is working. The conflict ⟨A, B⟩ is replaced by ⟨QA1, QX1, QO1, B⟩. Now [QA1], [QX1], [QO1], and [B] will be candidates. Steps 5 through 7: instantiate functional components all the way to the level of gates, within the full-adders fa1 and fa2. Suppose the conflict ⟨QA1, QX1⟩ is discovered. Now [QA1] and [QX1] are candidates. Step 8: split the assumption QA1; Step 9: split the assumption QX1. There are no instantiations to do, since the gates were primitives.

7.3 Ranking and Refinement

The ranking of candidates in XDE takes syndromes into account. The method is an extension of the candidate ranking method discussed in the previous chapter.

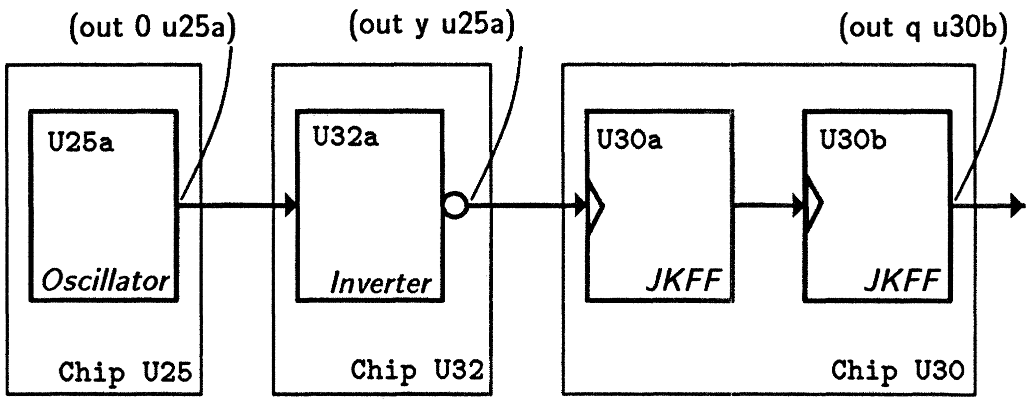
Without syndromes, candidate ranking works as follows. Each component is assigned a prior probability that it is working based on an estimate of its physical complexity. Assuming independence among failures in all components, the probability of a candidate is thus the probability that all components have just the status assigned in that candidate. For example, the candidate [U25] assigns the status "other" to U25 and "working" to the other components. The probability assigned this candidate is then $(1 - p(\text{U25})) \times p(\text{U30}) \times p(\text{U32})$. All candidates are then assigned a weight that is their probability normalized with respect to all the minimal candidates. This scheme yields intuitively satisfying results, since candidates involving single faults are generally more likely than those with multiple faults, and the candidates with the highest weights are those involving components with higher failure rates.

In XDE, components can have statuses other than simply working or not working, so there will be more candidates and a more elaborate ranking function. The benefit of the additional complexity and expense is that the troubleshooting engine exhibits better focusing. When candidates involving syndromes are shown to be inconsistent with observations, other candidates will appear more likely, and the troubleshooting engine will focus its efforts on those likelier candidates.

To use syndromes, XDE *refines* candidates by installing assumptions of the form "physical component X is exhibiting syndrome S," denoted X_S. For example, oscillator chips have the syndrome `inactive`; an assumption that oscillator U25 is inactive is denoted $U25_{Inactive}$. Because each of the statuses of a component are mutually exclusive, creating this assumption would result in the conflict $\langle U25_W, U25_{Inactive}\rangle$. The assumption that U25 is inactive results in the prediction that its output will have frequency zero, which in turn has other consequences. Usually, new conflicts involving $U25_{Inactive}$ will be discovered. Candidates are still constructed as the minimal covering sets of conflicts, but to deal with syndromes it is necessary to consider the complements of the candidates, the *maximal consistent environments.* A maximal consistent environment is one to which no assumption can be added without making it inconsistent. There is one maximal consistent environment per candidate. XDE constructs *diagnoses* from maximal consistent environments as illustrated by example below.

Consider a version of the clock generator troubleshooting example, shown in Figure 7.7. The three field replaceable components are the chips U25, U30,

Figure 7.7: Clock Generator



and U32. To better illustrate the refinement operation, assume that (i) chips are primitives and etches do not fail, and (ii) all antibehavior rules are disabled. The initial symptom that `(out q u30b)` is a constant `1` instead of having frequency 2.5 Mhz yields the conflict $\langle U25_W, U30_W, U32_W\rangle$, meaning that one of these components is faulty. Refining the candidate $[U25_W]$ with the syndrome $U25_{Inactive}$ yields the conflict $\langle U25_W, U25_{Inactive}\rangle$ as well.

$U25_{Inactive}$ is consistent with the observations and with U30 and U32 working properly. The minimal covering sets of these two conflicts are $[U25_W]$, $[U30_W,U25_{Inactive}]$, and $[U32_W,U25_{Inactive}]$. The maximally consistent environments are their complements $\{U30_W,U32_W,U25_{Inactive}\}$, $\{U25_W,U30_W\}$ and $\{U25_W,U32_W\}$ respectively. Each maximally consistent environment denotes a consistent assignments of statuses to every component.

These environments denote three possibilities: either (i) U30 and U32 are working and U25 is exhibiting syndrome `inactive`, or (ii) U25 and U30 are working and U32 has status `other`, or (iii) U25 and U32 are working and U30 has status `other`. There is a fourth possibility, that U30 and U32 are working and U25 has status `other` — it might be neither working nor inactive. Each maximal consistent environment that contains assumptions about syndromes yields several diagnoses, one for each subset of those assumptions. In this case there is only one such assumption and hence only one extra diagnosis. This yields four *diagnoses* in all, three corresponding to maximally consistent environments and one created by deleting assumptions about syndromes from those environments.

Each diagnosis that XDE generates in this manner specifies a single status for each component mentioned by any assumption. For brevity of notation, a diagnosis is denoted $[\![\ldots]\!]$ and shows only the component statuses that are not `working`. For example, $[\![U25_{Inactive}]\!]$ denotes a diagnosis in which only the assumptions $U30_W$, $U32_W$, and $U25_{Inactive}$ are present. $[\![U25_{Other}]\!]$ denotes a diagnosis in which only $U30_W$ and $U32_W$ are present.

Each diagnosis has an initial likelihood corresponding to the prior probability that every component has the status assigned, assuming independence between components[3]. The distribution assigned to each set of component statuses is derived from the physical complexity of the component, as described in Chapter 6. The weight assigned to each diagnosis is its likelihood

[3]Although BASIL guarantees that physical components do not share parts so that their failures can be assumed to be independent, XDE does handle the more general case of shared parts. Each maximal consistent environment may have several independent subsets of assumptions, each of which would derive the same consequences as the full environment. XDE computes the likelihood of diagnoses by taking the maximum likelihood of any independent subset, which is combinatorially expensive if independence is not maintained. Although not explored extensively, XDE should thereby be able to correctly assign likelihoods to diagnoses that involve dependent failures, since it would compute that likelihood based only on the likelihood of the original (independent) failure.

normalized over all diagnoses (U*nn* $_{\text{Other}}$ and U*nn* $_{\text{Inactive}}$ are hereafter abbreviated to U*nn* $_{\text{O}}$ and U*nn* $_{\text{I}}$):

Diagnosis		Likelihood	Weight
$[\![U25_I]\!]$	$p(U25_I) \times p(U30_W) \times p(U32_W) =$	.00984	.623
$[\![U30_O]\!]$	$p(U25_W) \times (1 - p(U30_W)) \times p(U32_W) =$	.00315	.200
$[\![U32_O]\!]$	$p(U25_W) \times p(U30_W) \times (1 - p(U32_W)) =$	.00276	.175
$[\![U25_O]\!]$	$(1 - p(U25_W) - p(U25_I)) \times p(U30_W) \times p(U32_W) =$	.000051	.00323

From the possible diagnoses, XDE now assigns a weight to each component based on the likelihood that it needs to be repaired. This is done by adding the weights of all diagnoses in which that component is faulty. For example, U25 is broken in both diagnoses $[\![U25_{\text{Inactive}}]\!]$ and $[\![U25_{\text{Other}}]\!]$, so both their weights contribute to the "repair weight" of U25. The table below shows the weights for U25, U30, and U32:

Component	Candidate Weights	Repair Weight
U25	.623 + .0032 =	.626
U30	.200 =	.200
U32	.175 =	.175

Continuing the diagnosis, suppose **`(11 (out y u32a))`** is observed to be changing. This yields the additional conflict $\langle U25_{\text{Inactive}}, U32_W \rangle$, since it is inconsistent for the oscillator to be inactive, the inverter working, and the output changing. In this case intuition says that the oscillator U25 is no longer as likely to be faulty; the new diagnoses shown below and their rankings support that intuition. The diagnoses involving U25 are much less likely than ones involving U32 or U30.

Diagnosis		Likelihood	Weight
$[\![U30_O]\!]$	$p(U25_W) \times (1 - p(U30_W)) \times p(U32_W)=$	.0032	.53
$[\![U32_O]\!]$	$p(U25_W) \times p(U30_W) \times (1 - p(U32_W))=$	.0028	.46
$[\![U25_O]\!]$	$(1 - p(U25_W) - p(U25_I)) \times p(U30_W) \times p(U32_W)=$	.000051	.0085
$[\![U25_I,U32_O]\!]$	$p(U25_I) \times p(U30_W) \times (1 - p(U32_W))=$	.000028	.0046
$[\![U25_O,U32_O]\!]$	$(1 - p(U25_W) - p(U25_I)) \times p(U30_W) \times (1 - p(U32_W))=$	.00000014	.000024

The repair weight associated with each component adjusts to the new ranking, indicating that U30 and U32 are much likelier to need repair than the oscillator U25:

Component	Candidate Weights	Repair Weight
U30	.53 =	.53
U32	.46 + .0046 + .000024 =	.46
U25	.0085 + .0046 + .000024 =	.013

The component status likelihood estimates can be perturbed greatly and still yield the same candidate rankings. It is the relative magnitudes of the likelihoods associated with statuses other than `working` in different components that matter, not their particular values. In the case of the clock generator, for example, the same rankings would have been obtained had the complexity of the oscillator been estimated as low as 40 (instead of 100), and as long as not all oscillator failures resulted in status `inactive`. The table below shows some examples of how much variation there can be. Each of the last four columns of the table below shows an alternative set of component status likelihoods that result in the same candidate rankings as above:

Component	Status	Likelihood			
U25	working	.60	.80	.999	.60
	inactive	.20	.15	.0005	.20
	other	.20	.05	.0005	.20
U30	working	.70	.85	.9998	.85
	other	.30	.15	.0002	.15
U32	working	.80	.90	.9999	.9999
	other	.20	.10	.0001	.0001

Note that the results remain stable even though likelihoods of the `other` and `inactive` statuses vary by orders of magnitude, so long as their order is preserved.

The scheme that XDE uses for generating and ranking diagnoses is expensive. Both GDE and XDE suffer from combinatorial explosion of candidates, but the refinement operation that XDE provides exacerbates the problem. In pathological cases the number of candidates (or maximal consistent environments) can be exponential in the number of conflicts, hence exponential in the number of components. In XDE, the number of candidates is at least exponential in the number of syndromes installed. Suppose there are n components C_i and each has one syndrome S. Then there are $2n$ assumptions, n of the form C_W and n of the form C_S. There are at least n conflicts $\langle C_W, C_S \rangle$.

These n conflicts share no assumptions and if there are no other conflicts, there will be at least 2^n candidates. In experiments with the current implementation of XDE, the amount of time each new refinement operation took approximately doubled and was stopped after the eighth refinement. Furthermore, there may be many maximal consistent environments containing more than one syndrome assumption. This has two undesirable consequences.

The first undesirable consequence is that a maximal consistent environment with n syndrome assumptions generates 2^n diagnoses, one for each combination of those n syndromes. For example, if it is consistent for X to have failure status S1 and Y to have failure status S2 simultaneously, then it is also consistent for X to have status S1 and Y to have status `other`, and vice versa. Thus one maximal consistent environment generates three diagnoses. Although several different maximal consistent environments may generate the same diagnoses, the potential for further combinatorial explosion is present. XDE does not do anything about this problem. It maintains the complete set of maximal consistent environments and diagnoses computable from the current conflicts.

The second undesirable consequence is that syndromes add new information to the behavior model from which many useless deductions will be made unless some additional control is exercised. Since syndromes usually have low likelihoods, environments containing multiple syndromes will have exceptionally low relative likelihoods. Each syndrome results in new predictions being made in the behavior model; for example, the `inactive` syndrome for oscillators results in the prediction that the frequency of the oscillator output is `0`. Since the predictions from different syndromes will interact, there will be many predictions that are present only in environments of very low likelihood. To deal with this problem, XDE controls the running of rules in such a way as to avoid doing work in environments of low likelihood. XDE pays the price of explicitly switching from one maximal consistent environment to the next, making predictions only in that one environment, and thereby only working on a few diagnoses at a time. This allows XDE to look for contradictions only in the diagnoses with the highest weights, never making deductions in environments whose likelihoods lie below a fixed threshold percentile. Explicit context switching is a high price to pay for this control, because the worst-case overhead is proportional to the total number of clauses times the number of diagnoses that get explored. However, it is possible to get the best of both worlds, and [deKleer86b] and [Geffner86] both demonstrate schemes upon which a more efficient implementation might be built someday.

To summarize, the procedure that XDE performs whenever a new conflict is discovered is as follows:

1. Update the set of maximal consistent environments. Maximal consistent environments are the complements of *candidates* as constructed by GDE.

2. Generate the set of diagnoses from the maximal consistent environments. Diagnoses are the subsets of the maximal consistent environments obtained by deleting syndrome assumptions.

3. Assign a probability to each diagnosis. Since each diagnosis assigns a status to every component mentioned by the universe of assumptions, the probability of the diagnosis is computed as the probability of the conjunction of all those statuses.

4. Normalize the probability of each diagnosis with respect to all the other diagnoses. This is the weight of each diagnosis.

5. Compute the repair weight of each component. The repair weight is the sum of the weights of diagnoses in which that component is broken.

If no syndromes are ever introduced, the set of diagnoses is the same as the set of maximally consistent environments, and the ranking is then exactly as in GDE. The addition of syndromes into that basic troubleshooting engine obviously introduces complexities into the generation and ranking of diagnoses. The advantage of doing so is that introducing a new syndrome assumption into an existing set of diagnoses can drastically shift the distribution of weights among the diagnoses, provided that the syndrome turns out to produce new conflicts with existing or subsequent observations. For example, in the clock generator used as an example throughout this section, without the syndrome $U25_{\textbf{Inactive}}$, the observation that `(11 (out y u32a))` was changing would have added no new information and the oscillator would have remained a likely diagnosis. With it, the weights of candidates involving U25 are all reduced below 2% each.

7.4 Making Observations

XDE selects informative observations using the same heuristic one-level lookahead strategy as GDE, but there are complications that arise in the digital circuit domain as represented in TINT. Among these complications are that (i) imprecise predictions hamper the ability of the lookahead strategy to make good choices, (ii) observations must be temporally quantified, and (iii) the possible observations have differing granularities and costs. XDE has partial solutions to the latter two problems, but the problems resulting from imprecise predictions are fundamental to any representation that trades precision for efficiency. After a brief review of the probe selection strategy, each of these issues will be considered in turn.

The expected information from a given observation can be quantified using the entropy of the possible outcomes of the observation. The entropy is the sum of $p_i \log p_i$ where i ranges over all outcomes and each p_i is the combined weight of the diagnoses that predict outcome i. Continuing the clock-generator example from above, the following set of diagnoses and their weights result after the initial symptom is discovered:

Diagnosis	Likelihood	Weight
$[\![\mathrm{U25_I}]\!]$	.00984	.623
$[\![\mathrm{U30_O}]\!]$	.00315	.200
$[\![\mathrm{U32_O}]\!]$	.00276	.175
$[\![\mathrm{U25_O}]\!]$	.000051	.00323

The behavior model makes many predictions; a small sample is shown below along with the environments in which they hold and the weights of the diagnoses that are definitely consistent with those environments. The prediction that the output of U30a is not changing is true in the empty environment and so is known to be consistent with all the diagnoses:

Signal	Value at time 10^6	Environments	Weights of Consistent Diagnoses
`(changing-wrt 0 10`^{6}` (ll (out 0 u25a)))`	t	$\{U25_W\}$	.200, .175
`(changing-wrt 0 10`^{6}` (ll (out 0 u25a)))`	nil	$\{U25_I\}$	.623
`(changing-wrt 0 10`^{6}` (ll (out y u32a)))`	t	$\{U25_W, U32_W\}$	.200
`(changing-wrt 0 10`^{6}` (ll (out y u32a)))`	nil	$\{U25_I, U32_W\}$	.623
`(changing-wrt 0 10`^{6}` (ll (out 0 u30a)))`	nil	$\{\}$	.623, .200, .175, .0051

Each of the ports **(out 0 u25a)**, **(out y u32a)**, and **(out 0 u30a)** can be observed to see whether its logic-level signal is changing. The expected benefit of making an observation at each port is the negative of the entropy of the distribution of weights among the various outcomes. An approximate version of the computation is shown in the table below.

Port	Sum over $-p_i \log p_i$	
(out 0 u25a)	$-(.200 + .175)\log(.200 + .175)$ $-(.623)\log(.623)$	$= .956$
(out y u32a)	$-(.200)\log(.200)$ $-(.623)\log(.623)$	$= .890$
(out 0 u30a)	$-1\log 1$	$= 0.0$

The last line shows that probing a signal that has already been observed has zero value. The other values indicate that probing the output of the oscillator **u25a** maximizes the expected information and so is preferable to other probes (when different probes yield the same estimated information XDE picks one of them essentially at random).

The relative likelihoods of component statuses **working**, **other**, and so forth impact the probe selections by influencing the weights of diagnoses. Diagnoses with high weights tend to bias XDE toward choosing probes in the vicinity of the components they mention. For example, had the likelihood of failure in U30 been greater than the likelihood of failure in the oscillator, the probe at **(out y u32a)** would have been chosen instead. Roughly, the higher the repair weight of a component (that is, the more diagnoses it appears in and the higher the relative likelihoods of those diagnoses) the more highly ranked the probes in its vicinity.

7.4.1 Prediction Strength and Probe Selection

Weak predictions of behavior cause the troubleshooting engine to make poor estimates of the information to be obtained at possible probe points. This in turn may cause it to wastefully ask for observations that do not produce any informative conflicts. As discussed earlier, there are several reasons why the behavior representation may be unable to make predictions: (i) abstractions may result in component behaviors not being total functions, (ii) local propagation of signal values may reach impasses, or (iii) the behavior of components may be too complex for there to be any good antibehavior rules. In the clock generator example being used at the moment, the reason is that the antibehavior rules have been disabled for presentation purposes.

Weak predictions raise the technical problem of estimating the expected information from a probe when some diagnoses make no prediction about the outcome of the probe. For example, $[\![U25_O]\!]$ makes no prediction about the signal at port `(out 0 u25a)`. The problem is that computing the entropy requires a distribution of probabilities that sum to 1. There are at least four ways of handling the weight that should be distributed among the diagnoses that make no prediction:

Assume that the other diagnoses predict some value that is different from all the explicitly predicted values. This is an optimistic assumption and tends to overestimate the information from a probe. For example, suppose that diagnoses carrying .5 of the weight predict that a particular signal will be changing, but the others make no prediction. The information .69 in this case would be computed the same as if all those other diagnoses had predicted the signal would not be changing. But suppose that diagnoses carrying weight .33 predict it will be changing, and others carrying .33 predict it will not. This method would estimate the information as 1.09, although there are really only two possible outcomes and the information cannot possibly be more than 1.

Assume that the other diagnoses predict the value that is likeliest among the possible values. This is a pessimistic assumption, tending to underestimate the information. For example, if diagnoses carrying .4 predict the signal is changing and others carrying .3 predict it is not, the result is computed as if the distribution had been .7 and .3, so the information is .61. If diagnoses carrying weight .5 predict a signal is changing and the rest make no prediction, the result is computed as if all diagnoses had predicted it would be changing too. Thus the information is 0.

Assume that the distribution of outcomes among the remaining diagnoses matches the distribution among the explicitly predicted outcomes. In general this provides more optimistic estimates than a method in which all possible outcomes are known, but an overly pessimistic estimate of 0 in the case where only one outcome has been explicitly predicted.

Assume that all possible outcomes are equally likely, and distribute the weight among them. This is the method used by GDE. Suppose for example that there are four possible outcomes a through d, with $p(a) = .3$, $p(b) = .2$, $p(c) = .1$, and $p(d) = 0$. This leaves a weight of .4, and this method yields a distribution of $p(a) = .4$, $p(b) = .3$, $p(c) = .2$ and $p(d) = .1$, and information of 1.28. The number of outcomes can be treated as $+\infty$ if not known.

This last method usually makes estimates that fall between those of the first and second methods above, and does not exhibit the anomalous behavior of the third when only one outcome has been explicitly predicted. It has other anomalies, however. Consider a signal `X` that is completely disconnected from the current set of candidates. No diagnosis predicts whether it is changing or not. According to this method, probing `X` is more informative than probing a signal `Y` that two-thirds of the diagnoses predict will be changing and that the other one-third predict will not.

XDE uses method 4 because it makes reasonable estimates and its principal anomaly is easy to avoid: signals for which no diagnosis predicts a value are never probed. The values XDE computes for each of the three probes are shown below. These are more accurate versions of the approximate values shown earlier, although the differences are very small and the relative rankings in this case have not changed.

Port	Sum over $-p_i \log p_i$	
`(out 0 u25a)`	$-(.200+.175+.0025)\log(.200+.175+.0025)$ $-(.623+.0025)\log(.623+.0025)$	$= 0.954$
`(out y u32a)`	$-(.200+.088)\log(.200+.088)$ $-(.623+.088)\log(.623+.088)$	$= 0.890$
`(out 0 u30a)`	$-1\log 1$	$= 0.0$

7.4.2 Temporal Quantification and Granularity

The behavior of a circuit can be observed at various times and at temporal granularities, with varying cost in setup time and difficulty. XDE currently has a simple and limited treatment of these issues.

Signals must be observed over time intervals. Each observation in XDE is a TINT `thru` predication and is part of some signal history. The expected information gain from the probing of any signal is the maximum for any interval during its history. Thus, when XDE suggests that (say) signal `(l1 X)` be probed, it means that there is some interval of its history during which an observation would be useful. XDE presents to the user the entire signal history of `(l1 X)` and abstractions of it along with some typical misbehaviors (a constant `1`, for example). The actual observations made of the device will probably correspond to one of the intervals already presented; if not, then an interval describing the observation can simply be typed in. For example, XDE may expect the value to be observed at a certain signal to be either 10 or 12, and so presents those as options; if the actual observation was 13 that can be typed in too. All observations are assumed to be completely accurate in terms of the signal values observed and the intervals over which they were seen.

The default interval over which signals are to be observed is denoted by a "global reference" timeline denoted by the pseudo-signal `GR`. The assertion `[thru ?a ?z GR t]` means that observations are made by default with respect to the time interval `?a` to `?z` inclusive. The interval `?a` to `?z` is referred to as the current "observation interval," which is automatically changed as the user adds new observations. The usual default is the ten second interval from 0 to 10^{10} nsec inclusive.

In a real troubleshooting session, the circuit board continues its behavior while the troubleshooter thinks about what to do next, and each new observation is made at a later time than the last. Since it would be unwise to assume that the circuit is not changing its state, the troubleshooter ordinarily forces it into a known state before making each new observation (by pressing a "reset" button, for example). The troubleshooter ordinarily further assumes that if the observations of the circuit are made more than once, the same results will be obtained each time. XDE has these assumptions built into it. For example, in troubleshooting the Audio Decoder each new observation is added over the interval from 0 to 10^{10}, rather than making each observation come after the previous one. Similarly, in the Input Encoder troubleshooting example, observations are added over the intervals $(-\infty, +\infty)$, $[1 \times 10^9, 2 \times 10^9]$, $[2 \times 10^9, +\infty)$, and $[0, 10^{10}]$, in that order. It is assumed that each new observation is made after pressing the reset button and providing identical test inputs to those before, so that the same behavior predictions are obtained.

Observations of different kinds of signals at different locations have different costs in setup time. Currently XDE only allows signals to be observed at the external ports of pins, where the pin meets the etch (although for clarity most of the examples elsewhere show observations being added at the closest port of some functional component). Observations also cannot be made over intervals shorter than one second. XDE associates a numerical cost with each possible probe, and its probe suggestions are biased to favor cheaper observations by multiplying the expected information of each probe times its cost. The costs currently used are as follows; they are estimates based on the relative ease of making the observation:

- Observing whether a logic-level signal is `1` or `0` all through the current observation interval costs 1.0. This is the most basic kind of observation and involves placing a single probe.

- Observing whether a logic-level signal is changing with respect to the current observation interval costs 0.9. This is slightly easier than viewing the actual value of the signal.

- Observing the swing of a voltage with respect to the current observation interval costs 0.9. Observing the amplitude of a signal is judged to have about the same difficulty as judging whether it is changing or not.

- Observing the frequency of a logic-level or voltage signal during the current observation interval costs 1.1, since it may require adjusting the temporal resolution of the oscilloscope.
- Observing the value of a signal sampled with respect to a clock costs 2.0, since it involves setting up two probes, one a strobe for the other.
- Observing the frequency of a two-phase clock signal costs 2.0, since it too involves setting up two probes.

Observing the outputs of the Input Encoder cost 1.0 no matter where they are physically located; these are assumed to be observable through other hardware not explicitly represented. The brightness of the console screen, for example, is an indirect way to observe the brightness signal.

7.5 Evaluation

Testing and diagnostic programs are usually evaluated by their *coverage* (the range of faults they can detect), *resolution* (the accuracy with which they can identify any fault actually present) and *speed* (as measured by the time it takes the running program to isolate the fault). The combined troubleshooting system of XDE, TINT, and BASIL can be evaluated this way too, although it is important to distinguish which subsystem is responsible for the quality achieved along each dimension. In model-based troubleshooting, coverage, resolution, and speed all depend critically on the ability to detect conflicts between the actual behavior of the device and its predicted behavior. XDE cannot do anything without those conflicts; if the model is too weak to produce predictions that are falsifiable by observations, then XDE will ask for many observations but make no progress toward isolating the fault. Thus the importance of the device representation far outweighs that of the troubleshooting engine.

7.5.1 Coverage

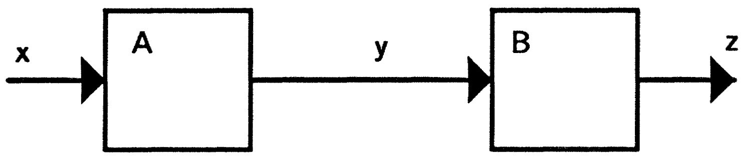
XDE needs to discover at least one discrepancy before it starts generating diagnoses. TINT, therefore, must represent enough detail about the behavior of the circuit as a whole to detect any misbehavior worth repairing. This does not imply that every misbehavior of every individual component needs to be detectable, although that is one way to guarantee coverage. For example, if the specifications of the Console Controller Board say that the screen brightness should increase in response to the "b" command, but do not specify how fast, then it is probably okay to represent that rate of change qualitatively instead of quantitatively. Any faults whose only effect would be to slow down the rate of advance would not be detected. The coverage provided by a behavior model is thus relative to the desired function of the whole device and of the detail of the observations.

The representation of the Console Controller Board in TINT is an incomplete prototype in this respect, since there are some functions of the board that its behavior definitions are too temporally coarse to represent. For example, if the board were faulty in such a way that large motions of the mouse across the table were to result in only small and sporadic motions on the screen, this would surely be considered a misbehavior. But since the TINT signals only represent the motion qualitatively it cannot describe the misbehavior. A rough measure of the coverage that the representation provides is to count the most common classes of faults, and determine which of them result in misbehaviors that can be distinguished. Among the most common faults are those that cause individual pins to act as open circuits. The Audio Decoder, for example, has nine chips having some 160 pins between them. Of these 160, failures in all but 30 would be detectable as discrepancies in the swing, frequency, and frequency in the first derivative of the voltage output of the digital-to-analog converter. Coverage of 80% of the common faults from only these three features of the output voltage is not bad, and would probably be improved with more detailed behavior rules for the shift registers and counters.

7.5.2 Resolution

A model-based troubleshooting program provides diagnostic resolution in proportion to the structural and behavioral detail that the device model provides. The program cannot of course distinguish between components that are not represented separately. BASIL, for example, represents an entire etch as a single component, so a break anyplace in the etch results in the same diagnosis. A subtler problem is that even failures in components represented separately cannot be distinguished if their behavior models and observations are insufficiently detailed. For example, Figure 7.8 shows a two-component device whose A and B components have the behaviors A and B. Suppose that x and z have been observed and a discrepancy detected at z. $\langle A_W, B_W \rangle$ is a conflict and the diagnoses are $[\![A_W]\!]$ and $[\![B_W]\!]$.

Figure 7.8: Distinguishing Between Diagnoses

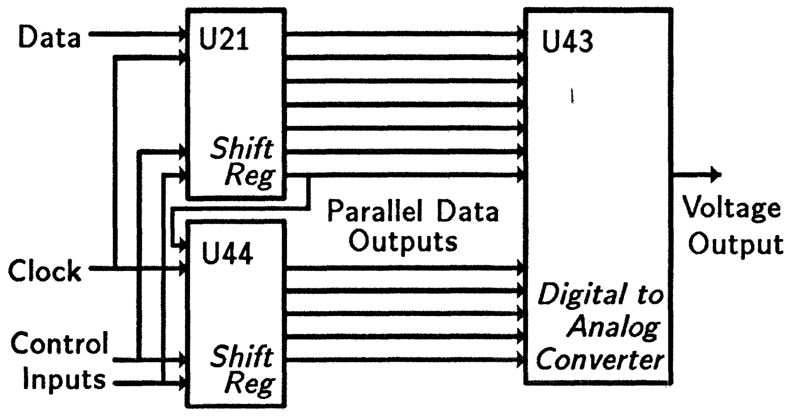


To distinguish between these diagnoses requires an observation at y — but it also requires that either the observation contradict (A x), or that (B y) contradict the observation at z. It might do neither. The observation might be too coarse, the behaviors might be partial, or both. There is nothing wrong with the troubleshooting engine; short of having an exhaustive set of syndromes for A or B, there is nothing it can do. The model is too weak.

The temporally abstract models of the Console Controller Board cause problems for XDE that are very much like this example. In principle TINT can represent the temporally detailed behavior in terms of logic-levels 0 and 1 for every gate in a circuit, and hence in principle XDE can detect misbehavior in any individual component. In practice, TINT rules only cover the temporally coarse behaviors that are easy to observe. For example, an open circuit on a control input of the shift register U21 might result in all its parallel data output signals changing, although in seemingly random fashion that would show up on the digital-to-analog converter (U43) output as such (Figure 7.9).

Even assuming that every visible node in the Audio Decoder were probed to see whether it was changing or not, there would nevertheless be no way to distinguish between the diagnoses $[\![U21_{\mathrm{Other}}]\!]$ and $[\![U43_{\mathrm{Other}}]\!]$. The outputs of U21 are not represented in enough temporal detail for a discrepancy to be detected there. More generally, among the 130 detectable common faults in the Audio Decoder, about half of them are distinguishable down to a single chip and the remainder result in this kind of ambiguity. Beyond the common faults, it is probably the case that most faults internal to the chips would result in similar lack of resolution. The temporal detail of the predictions is

Figure 7.9: Detail of Audio Decoder



sufficient to allow many correct diagnoses but is insufficient to achieve perfect resolution, even with exhaustive probing. Ultimately, given any particular level of structural detail, if perfect resolution is desired there will always be cases that require detailed timing information.

7.5.3 Speed

An appropriate measure of the speed of a model-based troubleshooting program is the number and cost of the observations it requires to reach its final diagnosis. This is a meaningful measure so long as the device model provides enough resolution that there is in fact such a thing as a "final diagnosis." If the behavior model is too weak or the observations too coarse to distinguish different components, XDE eventually quits after asking for all possible observations. The speed metric in that case is hardly meaningful. Even if the model and observations do provide sufficient detail to discriminate components down to the primitive level of detail, the model may still be too weak to discover genuine conflicts between what has been observed and what should have been. In that case more observations will be required than strictly necessary. The probe selection strategy used by XDE has a number of heuristic aspects: (i) it is influenced by component failure rates that are estimates, (ii) it estimates the benefits of probes with a one-level lookahead rather than searching through all possible sequences of observations, and (iii) it estimates information from probing signals whose predicted value is not known in all diagnoses by assuming that all observation outcomes are equally likely. However, no matter how good these heuristics are, in the long run their positive impact on the probes actually chosen are unlikely to be nearly as strong as the negative impact of a device model that cannot make full use of the observations actually chosen. The cleverest strategy for choosing observations cannot make up for observations and models that are too coarse to detect discrepancies.

7.6 Summary

The model-based troubleshooting engine XDE extends GDE by incorporating hierarchic diagnosis and fault models. Hierarchic diagnosis is achieved with the *decomposition* operation, which descends one level at a time through both the physical and functional hierarchies in BASIL. Knowledge about how components fail, represented as *syndromes*, is used in the *refinement* operation. Syndromes help focus the troubleshooting process by biasing the suggestion of new observations away from components unlikely to be failing. XDE can suggest observations of signals at various temporal resolutions, and it biases its suggestions toward those that are cheaper.

Like all model-based troubleshooting engines, XDE is almost totally dependent on the device model and on the technology for observing the real device. Obviously, if the model lacks fidelity its diagnoses may be incorrect. A subtler problem is that if the model is imprecise — if it fails to produce falsifiable predictions — the troubleshooting engine will be indiscriminate, never reaching a conclusive diagnosis no matter how many observations are made. In light of this dependence, any evaluation of the quality of the diagnoses that XDE produces is really an evaluation of the quality of the underlying device model.

Chapter 8

Conclusions and Future Work

Model-based troubleshooting has not previously scaled up to deal with complex devices such as digital circuit boards. This is because traditional analytic models of complex devices do not explicitly represent aspects of the device that are important for troubleshooting. This report has described a digital circuit representation that was constructed with troubleshooting explicitly in mind, a representation that enables the general model-based troubleshooting engine XDE to successfully diagnose failures in circuits that are much more complex than any previously attempted. This representation is embodied in the language BASIL for representing the physical and functional organization of circuits and in the temporal reasoning system TINT for representing circuit behavior. The modeling principles that underly these languages and govern their use concern ways in which features of the circuit relevant to troubleshooting can be made explicit:

- Components in the representation of the physical organization of the circuit should correspond to the possible repairs of the actual device.

Making the elements of the structure representation correspond to possible repair actions ensures that the troubleshooting program will not waste effort trying to discriminate between diagnoses that have identical repairs. BASIL represents circuits using a strict hierarchy of physical components that reflects the way the board was manufactured and hence those parts that can be replaced.

- Components in the representation of the functional organization of the circuit should facilitate behavioral abstraction.

The only role that an explicit representation of functional organization plays in model-based troubleshooting is to make behavior prediction more efficient. In extracting the functional organization from a raw schematic the modeler need only represent what will make the behavior easier to reason with, rather than necessarily representing what the designer had in mind. BASIL represents this functional organization using a nonstrict component hierarchy whose leaves are shared with the physical hierarchy. XDE does hierarchic diagnosis using the physical and functional hierarchies by descending primarily through the physical hierarchy, while reasoning about the behavior of functional components roughly corresponding to each level of physical detail.

- The behavior of components should be represented in terms of features that are easy for the troubleshooter to observe.

Some features of time-varying signals are easier to observe than others. In digital circuits, temporally coarse features of signals are easier to observe than clock-cycle-by-clock-cycle behavior. TINT provides a framework in which both abstractions and behaviors are functions from signals to signals, along with a vocabulary of temporal abstractions including concepts such as *change*, *count*, and *frequency*. Expressing the behavior of components in these terms makes prediction more efficient while largely retaining the ability to detect the effects of common faults.

- The behavior of a component for which changes on its inputs always results in changes on its outputs should be represented in temporally coarse terms.

Given a set of temporal (or any other) abstractions, it is an interesting and relevant question to ask: for what class of behaviors it is possible to formulate easily computable and strong abstract behaviors? More specifically, given the language TINT and its vocabulary of temporal abstractions, for which components is it worth writing temporally abstract behavior rules for? In the case of temporal abstractions, the natural class of relevant behaviors are those for which changes on inputs always result in changes on outputs. Combinational behaviors expressible as one-to-one functions, as well as toggles, counters, and shift registers, fall in this category.

- A temporally coarse behavior description that only covers part of the behavior of a component is better than not covering any at all.

Although the full behavior of a component may be too complex to reduce to a simple relationship between (say) the number of changes on its inputs and the number of changes on its outputs, there may be a useful relationship that involves only a subset of its inputs, assuming that the others are held constant. Similarly, there may be a useful relationship between different signals sampled with respect to a common clock. TINT rules for describing the temporally abstract behaviors of components ranging from boolean gates to microprocessors capture the normal behavior of those components using these techniques.

- A sequential circuit should be encapsulated into a single component to enable the description of its behavior in a temporally coarse way.

Although the individual behaviors of the components in a sequential circuit may not lend themselves to temporally coarse descriptions, the group may be performing a simple function when taken as a whole. Encapsulating the group of components makes it possible to apply other temporal abstraction techniques such as holding inputs constant. In many troubleshooting situations, it will be unnecessary to ever consider the individual state transitions of its sequential behavior.

- An explicit representation of a given component failure mode should be used if the underlying failure has high likelihood.

Components break in the field in certain ways much more often than in other ways. XDE takes advantage of this knowledge by extending the multiple-faults approach of GDE [deKleer87] to use fault models. The notion of a *syndrome* in BASIL and TINT captures knowledge about the likelihood, physical causes, and local behavioral effects of failures. Syndromes are beneficial when they are inconsistent with the symptoms, since this can reduce the ambiguity among the possible diagnoses. A syndrome with relatively high likelihood is valuable because it can be used to virtually eliminate an otherwise logically possible diagnosis.

- An explicit representation of a given component failure mode should be used if the resulting misbehavior is drastically simpler than the normal behavior of the component.

If a component with normally complex behavior has some potential internal fault or faults that cause it to misbehave catastrophically, then any partially correct behavior observed for the component makes it a less likely suspect. Syndromes that simplify the behavior of a component are useful because their effects on the rest of the device are relatively efficient to predict.

The power of these eight principles has been demonstrated in an implemented program that can troubleshoot problems in a board-scale circuit, the Symbolics 3600 Console Controller Board. Testing the system on a wider set of cases using the same board and modeling yet other boards are among the important follow-up work that needs to be done. The following sections discuss three avenues of further research that should be performed: (i) improving the engineering of the program, (ii) deriving the specialized troubleshooting representation from more primitive circuit descriptions, and (iii) generalizing the methodology beyond the domain of digital circuits.

8.1 Engineering Issues

The current implementations of XDE, BASIL, and TINT are demonstration vehicles and are too slow to contemplate putting to serious use in troubleshooting. Since their implementation details have not been presented in this report it would be inappropriate to discuss in detail how those implementations could be improved, nevertheless it is worth mentioning certain broad areas needing improvement.

Circuit structures are described in BASIL using the predicates **`isa`**, **`ako`**, **`has-port`**, **`conn`**, **`status-of`**, and **`corr`**, and assertions using these predicates are stored in the most naive and general fashion as patterns in a discrimination net. A better implementation would store the components and connections as frame instances [Minsky75] [Batali81] [Davis83]. JOSHUA provides a substrate for doing so [Rowley87], but the conversion has not yet been done. Moreover, while building the BASIL description of the Console Controller Board the lack of graphical display and editing facilities was keenly felt; reimplementing BASIL using an existing design and layout language might be no more difficult than using any arbitrary frame language while yielding considerably more utility.

TINT is slow in spite of its simplicity. As with BASIL, assertions involving the predicates `thru` and `tsame` are implemented (primarily) as patterns in a discrimination net, and this generality is costly. A deeper problem is that during the prediction process the forward chaining from these assertions makes many deductions about time intervals that later turn out to be shadowed. This problem might be alleviated if assertions were made about time intervals with relative endpoints instead of fixed integers. The predecessor to TINT was a temporal constraint propagator MINT that used inequalities over time points in that fashion. Goal-directed reasoning about these inequality constraints was integrated with the forward chaining from intervals of signal histories. However, the effort to give this more complex program adequate performance turned into a major research agenda all its own and was suspended in favor of the simpler TINT ontology. The next incarnation of the temporal reasoning subsystem will probably not be a data-driven constraint propagator at all, but a goal driven system that produces more limited predictions.

The hybrid TMS used in the program is basically a single-context TMS to which the propagation of environments and labels has been added. Labeling each assertion with its minimal environments is very useful for probe selection, implying the need for an Assumption-based TMS architecture (ATMS). On the other hand there are at least two reasons for doing explicit context switching: (i) TINT requires that some assertions be "shadowed" to prevent rules from firing on them, an effect that seems difficult to produce in an ATMS, and (ii) unrestricted rule firing in environments containing several syndromes would be wasteful since the relative likelihood of those environments is usually very small. An ATMS that provided shadowing and efficient incremental updating of environment likelihoods (so as to support best-first search among diagnoses) would be a good replacement for the current hybrid TMS.

XDE currently tries to use fault models, then tries to do hierarchic diagnosis, and when those fail it selects probes. A better strategy would make use of the number of outstanding diagnoses and ambiguity among the diagnoses to choose the next operation. For example, when there are many diagnoses, getting new observations is probably preferable to doing decompositions. Experiments with a strategy based on the entropy of the current set of diagnoses did not yield significant improvement, but better control is clearly necessary. The decomposition operation in particular is invoked far too casually, resulting in many useless predictions being made.

Finally, XDE spends a surprising amount of time finding optimal probes[1]. The basic reason is that the amount of work involved each time a probe is chosen is proportional to the number of possible probes times the number of candidates. However, since the only interesting probe is the one having minimum entropy, there ought to be a way of sorting the possibilities so that not every candidate or probe needs to be examined every time. Also, not every candidate likelihood changes between observations, so there should be some way to cache parts of the computation from one probe selection to the next.

8.2 Deriving the Representation

The fact that the behavior rules for the components are currently all hand-crafted is a cause for concern. In the short term, a library of signal definitions, behavior constraints, and syndromes has been accumulated to speed up the description of future circuits. However, the whole process needs to be automated: the troubleshooting program should be able to diagnose a circuit starting only from a primitive representation of structure and part specifications along with whatever design specifications and annotations happen to be available for its various modules. Presumably this would be done by building and using an intermediate representation of its structure and behavior along the lines described in this report.

Extracting an appropriately abstracted behavior representation from an underlying physical structure is an exceptionally difficult problem. Since the appropriate abstractions to be used for describing circuit behavior are bound to capture some of the intended function of the circuit, there are close connections between this and the function-from-form problem. [deKleer78] presents as a key insight a teleological constraint: the correct interpretation of the function of a designed artifact must assign some role to every structural element. A latent flaw in that particular approach was that the target representation of circuit function seemed to exist in a vacuum, having no role in any problem solver. In FUNSTRUX [Hall87], by contrast, simulation models

[1]So does GDE (B. Williams, personal communication).

of digital circuit elements are symbolically composed into simulation models for aggregate structures; the compositions and subsequent simplifications are strongly guided by the goal of producing efficient simulation models for a specific event driven simulator. In the present case, the target problem solver is XDE and so the desirable characteristics of the target representation are clear. This should provide a strong constraint on the relevant abstractions. The FUNSTRUX approach might also work under the somewhat different goal of producing temporally abstract behaviors. Finally, since the image of an ordinary digital model under temporal abstractions can be viewed as a reformulation into a specialized representation, the frameworks outlined in [Kramer87] or [VanBaalen88] might be useful ways of approaching the problem.

An alternative approach would be to start doing prediction at low levels of detail, but recognize recurring patterns of events and extrapolate their cumulative effects over large stretches of time. This is the essence of the *aggregation* technique [Weld86]. There are at least two difficulties with this approach: (i) recognizing what constitutes a "recurring" pattern of events, that is, deciding which events are relevant to a given pattern, and (ii) ensuring that the extrapolated predictions are robust against fencepost errors. In spite of these difficulties it bears further investigation because it has strong intuitive appeal — people seem to be good at detecting repetitive sequences and extrapolating them to find their limits. At the very least, aggregation should be a useful technique for generating fault syndromes from ordinary fault simulations.

8.3 Generalizing the Methodology

Digital circuit troubleshooting is a relatively narrow domain. To learn more about model-based troubleshooting of complex systems in general it is important to apply the technology to systems in a variety of domains. The eight principles of modeling for troubleshooting that form the core of this work are briefly discussed below in the context of local area computer networks, automobile engines, and internal medicine.

The eight principles can be used to suggest characteristics of a representation for troubleshooting computer networks. First, there is a superficially appealing analogy to be drawn between the structure of computer boards and the structure of networks. Instead of chips connected by wires, there are hosts connected by cables, and so forth. However, one of the principles dictates that the elements of the structure should correspond to failures and repairs. On closer investigation of the domain it turns out that failures and repairs in networks only rarely have a physical cause such as a broken cable; the most typical failures are crashed server processes or operating systems, for which the repair involves a restart operation. This indicates that, for example, components such as hosts are not appropriate physical primitives, but that in some sense server processes are. Second, tests are usually designed into the system and can be quite cheap. Some networks, for example, provide an operation that allows one host to request a status response from every host on its subnetwork. To model the features of component behaviors that are easiest to observe means that for the most part only the behavior of the hosts with respect to these test operations needs to be modeled. Finally, there are other network misbehaviors for which the principal symptoms are temporally coarse — mail servers, for example, are notorious for building up enormous queues that ultimately result in slowed response times. This suggests that appropriate behavior models will deal not with the movements of individual packets, but rather with temporally abstract features such as the number of packets and their rates of transmission. All of the principles for constructing temporally abstract behaviors appear to apply equally well to network events as to digital events.

There are few obvious analogies between automobile engines and digital circuits, but from the special perspective of troubleshooting and the principles of modeling for troubleshooting engines have important similarities to circuits. First, they are manufactured artifacts that are repaired by replacement of their physical parts. Second, the easily observed features of the engine behavior are temporally coarse compared to events such as piston firings and crankshaft rotations. Some of these temporally coarse features describe the behavior of subsystems with sequential feedback: for example, the distributor, pistons, crankshaft, and generator form a feedback loop whose interesting properties are its revolutions per minute, sputters, vibrations, and so forth. Third, there are many failure modes worth modeling explicitly either because they are common or because they drastically simplify the behavior of the engine: empty gas tanks, dead batteries, disconnected wires, and so forth. While it would surely be a major task to construct a full-blown model of an internal combustion engine, the eight principles do suggest which properties of engines will be most important to include in a model to be used for troubleshooting.

The methodology and principles in this work are most appropriate for troubleshooting designed artifacts. An implicit assumption has been that the modeler could in principle provide an arbitrarily detailed account of the behavior of the system, while the modeling challenge is to make do with the least detailed description that still works. This assumption does not apply to human physiology and medicine; the challenge in these domains is to produce *any* model. From the perspective of this work there are other important differences as well. First, it is inappropriate to emphasize the representation of physical structure. In medicine it is relatively rare that therapy consists of physically isolated structural repairs (organ transplants notwithstanding). Second, in medicine easily-observed symptoms are uncorrelated with their temporal extent. While it may be important to explicitly model what can be observed it generally has nothing to do with temporal abstractions. Third, the criteria used to decide which circuit fault models to include are at best incomplete for human diseases. For example, the short and long term seriousness of the diseases should somehow be taken into account. A few of the principles of modeling for troubleshooting might apply to subdomains of medicine for which good analytic models exist, but only tangentially. For example, in the multilevel physiological model in ABEL [Patil81], one of the simplifications that distinguishes the abstract levels from the detailed ones is that feedback loops are composed and summarized. In general, however, there is as yet no compelling evidence that the principles of modeling for troubleshooting will apply to modeling physiology for diagnosis.

Appendix A

Scenario Transcripts

The transcripts in Appendices A.1 through A.11 have three kinds of entries:

- "`There are` n `diagnoses...`" indicates the status of the troubleshooting engine after each change to its set of diagnoses. The current top diagnosis is shown with it.
- "`Adding observation...`" means that a new TINT assertion about the value of some observable signal is being added.
- "`Entropy     Signal; Aliases...`" shows the top ranked probe — its entropy, the internal name of the signal, two other nearby named ports to help provide context, and finally the list of values predicted there along with their labels.

To produce the transcripts, the troubleshooting engine consulted an oracle to get the result of its highest ranked probe, just as if a human user had typed in the same result. Some of the transcripts have a histogram at the end that summarizes the sequence of probes made. The length of each horizontal bar corresponds to the number of competing diagnoses. The bracketed timestamps `[hh:mm:ss]` give a rough idea of the performance of the troubleshooting program running on a Symbolics 3650 with 2 Mw; 16 minutes for one of the Audio Decoder examples is typical.

A.1 Clock Generator Example

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[19:03:19] Adding observation of 1 at
 [LL [HOLE 1 N167]]
...
Conflict!  There are 3 diagnoses (entropy 1.303) accounting for .95:
0.639 [[(U25 Other)]]
...
There are 3 diagnoses (entropy 1.303) accounting for .95:
0.639 [[(U25 Other)]]
...
Refining U25 with OPEN
...
There are 3 diagnoses (entropy 1.305) accounting for .95:
0.636 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U25 WORKING]>)
...
There are 3 diagnoses (entropy 1.305) accounting for .95:
0.636 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U30 WORKING]>)
...
There are 3 diagnoses (entropy 1.305) accounting for .95:
0.636 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U32 WORKING]>)
...
There are 3 diagnoses (entropy 1.305) accounting for .95:
0.636 [[(U25 Open)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.9431 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N291]]]
        aka [PIN 8 U25] aka [OUT 0 U25A]
        ((NIL #<ENV 2 06> #<ENV 1 010>) (T #<ENV 1 01>))
...
[19:05:25] Adding observation of T at
 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N291]]]
...
There are 2 diagnoses (entropy 0.997) accounting for .95:
0.533 [[(U30 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.9968 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N205]]]
        aka [PIN 2 U32] aka [OUT Y U32A]
        ((T #<ENV 1 02>) (NIL #<ENV 1 04>))
```

```
...
[19:06:06] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N205]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U32 Other)]]
...
```

A.2 Audio Decoder Example I

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[14:06:13] Adding observation of 0 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U12 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8291 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[14:10:00] Adding observation of 1 at
 [LL [HOLE 2 N290]]
...
Conflict!  There are 6 diagnoses (entropy 2.556) accounting for .95:
0.222 [[(U12 Other)]]
...
```

```
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U10 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U11 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U20 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N88]]]
        aka [PIN 14 U21] aka [BI 3 U21A]
        ((T #<ENV 3 0320>))
...
[14:12:03] Adding observation of 1 at
 [LL [HOLE 1 N88]]
...
There are 2 diagnoses (entropy 0.921) accounting for .95:
0.663 [[(U12 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.9164 [CHANGING-WRT 0 10000000000 [LL [HOLE 4 N56]]]
        aka [PIN 2 U11] aka [IN CLK U11A]
        ((T #<ENV 1 0200>))
...
[14:12:43] Adding observation of 1 at
 [LL [HOLE 4 N56]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U12 Other)]]
...
Probes  Diagnoses
(Four)  afterwards
  N272  ########## 10
  N290  ##### 5
   N88  ## 2
   N56  # 1
T
```

A.3 Audio Decoder Example I with Syndromes

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[14:26:10] Adding observation of 0 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U12 with INACTIVE
...
Conflict!  There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U11 with CSB-INACTIVE
...
Conflict!  There are 11 diagnoses (entropy 3.320) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 11 diagnoses (entropy 3.320) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U10 with CSB-INACTIVE
...
Conflict!  There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
```

```
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8293 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[14:30:37] Adding observation of 1 at
 [LL [HOLE 2 N290]]
...
Conflict!  There are 8 diagnoses (entropy 2.854) accounting for .95:
0.156 [[(U12 Other)]]
...
Conflict!  There are 6 diagnoses (entropy 2.534) accounting for .95:
0.203 [[(U12 Other)]]
...
There are 6 diagnoses (entropy 2.534) accounting for .95:
0.203 [[(U12 Other)]]
...
Refining U20 with CSB-INACTIVE
...
Conflict!  There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U12 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U10 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U11 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U20 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
```

```
...

...
Entropy          Signal; Aliases; Value-Environment Pairs
 1.0000 [LL [HOLE 1 N213]]
        aka [PIN 15 U10] aka [OUT TC U10A]
        ((0 #<ENV 1 04> #<ENV 3 04102> #<ENV 3 020102>))
...
[14:34:13] Adding observation of 0 at
 [LL [HOLE 1 N213]]
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...

...
Entropy          Signal; Aliases; Value-Environment Pairs
 1.0000 [LL [HOLE 1 N56]]
        aka [PIN 19 U12] aka [OUT Z U12A]
        ((0 #<ENV 2 04100> #<ENV 2 010100> #<ENV 2 020100>))
...
[14:35:12] Adding observation of 1 at
 [LL [HOLE 1 N56]]
...
There are 2 diagnoses (entropy 0.881) accounting for .95:
0.701 [[(U12 Other)]]
...

...
Entropy          Signal; Aliases; Value-Environment Pairs

Probes  Diagnoses
(Four)  afterwards
  N272  ########### 11
  N290  ###### 6
  N213  ###### 6
   N56  ## 2
T
```

A.4 Audio Decoder Example II

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[15:08:47] Adding observation of 0 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U12 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...

...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8291 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[15:12:34] Adding observation of 1 at
 [LL [HOLE 2 N290]]
...
Conflict!  There are 6 diagnoses (entropy 2.556) accounting for .95:
0.222 [[(U12 Other)]]
...
```

```
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U10 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U11 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U20 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N236]]]
        aka [PIN 14 U44] aka [BI 3 U44A]
        ((T #<ENV 4 0360>))
...
[15:14:42] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N236]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N117]]]
        aka [PIN 5 U44] aka [BI 4 U44A]
        ((T #<ENV 4 0360>))
...
[15:15:57] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N117]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N208]]]
        aka [PIN 24 U43] aka [IN 7 U43A]
        ((T #<ENV 3 0320>))
...
[15:17:06] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N208]]]
...
```

```
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
        aka [PIN 15 U21] aka [BI 5 U21A]
        ((T #<ENV 3 0320>))
...
[15:18:00] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
        aka [PIN 28 U43] aka [IN 11 U43A]
        ((T #<ENV 3 0320>))
...
[15:18:51] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5619 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N260]]]
        aka [PIN 8 U20] aka [IN A U20C]
        ((T #<ENV 1 0200>))
...
[15:19:42] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N260]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5619 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N232]]]
        aka [PIN 11 U43] aka [IN WR U43A]
        ((T #<ENV 3 01210>))
...
[15:20:34] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N232]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
```

```
...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.4434 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N213]]]
        aka [PIN 3 U20] aka [IN B U20A]
        ((NIL #<ENV 1 04>))
...
[15:21:28] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N213]]]
...
Conflict!  There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U20 Other)]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U20 Other)]]
...
Probes  Diagnoses
(Ten)   afterwards
  N272  ########## 10
  N290  ##### 5
  N236  ##### 5
  N117  ##### 5
  N208  ##### 5
  N289  ##### 5
   N48  ##### 5
  N260  ##### 5
  N232  ##### 5
  N213  # 1
T
```

A.5 Audio Decoder Example II with Syndromes

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[14:48:45] Adding observation of 0 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U12 with INACTIVE
...
Conflict!  There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U11 with CSB-INACTIVE
...
Conflict!  There are 11 diagnoses (entropy 3.320) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 11 diagnoses (entropy 3.320) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U10 with CSB-INACTIVE
...
Conflict!  There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
```

```
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 11 diagnoses (entropy 3.282) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8293 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[14:53:23] Adding observation of 1 at
 [LL [HOLE 2 N290]]
...
Conflict!  There are 8 diagnoses (entropy 2.854) accounting for .95:
0.156 [[(U12 Other)]]
...
Conflict!  There are 6 diagnoses (entropy 2.534) accounting for .95:
0.203 [[(U12 Other)]]
...
There are 6 diagnoses (entropy 2.534) accounting for .95:
0.203 [[(U12 Other)]]
...
Refining U20 with CSB-INACTIVE
...
Conflict!  There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U12 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U10 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U11 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U20 WORKING]>)
...
There are 6 diagnoses (entropy 2.532) accounting for .95:
0.218 [[(U12 Other)]]
```

```
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 1.0000 [LL [HOLE 1 N213]]
        aka [PIN 15 U10] aka [OUT TC U10A]
        ((0 #<ENV 1 04> #<ENV 3 04102> #<ENV 3 020102>))
...
[14:56:47] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N213]]]
...
Conflict!  There are 1 diagnoses (entropy 0.010) accounting for .95:
0.993 [[(U20 Other)]]
...
There are 1 diagnoses (entropy 0.010) accounting for .95:
0.993 [[(U20 Other)]]
...
Probes   Diagnoses
(Three)  afterwards
   N272  ########### 11
   N290  ###### 6
   N213  # 1
T
```

A.6 Audio Decoder Example III

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[11:00:23] Adding observation of 30 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[11:00:32] Adding observation of 2000.0 at
 [FWW 50000000 '(NIL T) [CROSS (EXPT 2 11) [VOLTAGE [HOLE 1 N272]]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[11:00:37] Adding observation of 20000.0 at
 [FWW 50000000 '(NIL T) [CROSS 0 [DT [VOLTAGE [HOLE 1 N272]]]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U12 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
```

```
...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8291 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[11:04:42] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7617 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N280]]]
        aka [PIN 10 U10] aka [IN ENBT U10A]
        ((T #<ENV 5 0307>))
...
[11:05:43] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N280]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7617 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N129]]]
        aka [PIN 11 U10] aka [OUT 3 U10A]
        ((T #<ENV 5 0307>))
...
[11:06:42] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N129]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7288 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N112]]]
        aka [PIN 21 U43] aka [IN 4 U43A]
        ((T #<ENV 4 0360>))
...
[11:07:43] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N112]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
```

```
 0.5980 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N88]]]
        aka [PIN 14 U21] aka [BI 3 U21A]
        ((T #<ENV 3 0320>))
...
[11:08:43] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N88]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5980 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
        aka [PIN 28 U43] aka [IN 11 U43A]
        ((T #<ENV 3 0320>))
...
[11:09:43] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5830 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N232]]]
        aka [PIN 8 U22] aka [OUT Y U22C]
        ((T #<ENV 3 01210>))
...
[11:10:43] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N232]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.4270 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N159]]]
        aka [PIN 19 U43] aka [IN 2 U43A]
        ((T #<ENV 2 0140>))
...
[11:11:45] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N159]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.4270 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N208]]]
        aka [PIN 6 U21] aka [BI 2 U21A]
        ((T #<ENV 2 0120>))
...
[11:12:46] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N208]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4270 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N194]]]
        aka [PIN 20 U43] aka [IN 3 U43A]
        ((T #<ENV 2 0140>))
...
[11:13:47] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N194]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4270 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N131]]]
        aka [PIN 22 U43] aka [IN 5 U43A]
        ((T #<ENV 2 0120>))
...
[11:14:49] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N131]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4270 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N139]]]
        aka [PIN 23 U43] aka [IN 6 U43A]
        ((T #<ENV 2 0120>))
...
[11:15:52] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N139]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4270 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N246]]]
        aka [PIN 5 U21] aka [BI 4 U21A]
        ((T #<ENV 2 0120>))
...
```

```
[11:16:53] Adding observation of 1 at
 [LL [HOLE 1 N246]]
...
Conflict!  There are 2 diagnoses (entropy 0.918) accounting for .95:
0.667 [[(U21 Other)]]
...
There are 2 diagnoses (entropy 0.918) accounting for .95:
0.667 [[(U21 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.6498 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N236]]]
        aka [PIN 14 U44] aka [BI 3 U44A]
        ((T #<ENV 2 0140>))
...
[11:17:44] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N236]]]
...
There are 2 diagnoses (entropy 0.918) accounting for .95:
0.667 [[(U21 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.6498 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N117]]]
        aka [PIN 5 U44] aka [BI 4 U44A]
        ((T #<ENV 2 0140>))
...
[11:18:48] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N117]]]
...
There are 2 diagnoses (entropy 0.918) accounting for .95:
0.667 [[(U21 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.6498 [CHANGING-WRT 0 10000000000 [LL [HOLE 5 N140]]]
        aka [PIN 10 U11] aka [IN ENBT U11A]
        ((NIL #<ENV 1 0100>))
...
[11:19:30] Adding observation of 1 at
 [LL [HOLE 5 N140]]
...
There are 2 diagnoses (entropy 0.918) accounting for .95:
0.667 [[(U21 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.6498 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N254]]]
        aka [PIN 9 U21] aka [IN CLEAR U21A]
        ((NIL #<ENV 1 0100>))
```

```
...
[11:20:14] Adding observation of 1 at
 [LL [HOLE 3 N254]]
...
Conflict!  There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U21 Other)]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U21 Other)]]
...
Probes    Diagnoses
(Twenty)  afterwards
    N272  # 1
    N272  ########## 10
    N290  ########## 10
    N280  ########## 10
    N129  ########## 10
    N112  ########## 10
     N88  ########## 10
     N48  ########## 10
    N232  ########## 10
    N159  ########## 10
    N208  ########## 10
    N194  ########## 10
    N131  ########## 10
    N139  ########## 10
    N246  ## 2
    N236  ## 2
    N117  ## 2
    N140  ## 2
    N254  # 1
T
```

A.7 Audio Decoder Example III with Syndromes

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:30:04] Adding observation of 30 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:30:12] Adding observation of 2000.0 at
 [FWW 50000000 '(NIL T) [CROSS (EXPT 2 11) [VOLTAGE [HOLE 1 N272]]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:30:17] Adding observation of 20000.0 at
 [FWW 50000000 '(NIL T) [CROSS 0 [DT [VOLTAGE [HOLE 1 N272]]]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U12 with INACTIVE
...
Conflict!  There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U11 with CSB-INACTIVE
...
Conflict!  There are 11 diagnoses (entropy 3.320) accounting for .95:
0.163 [[(U43 Other)]]
...
Conflict!  There are 10 diagnoses (entropy 3.209) accounting for .95:
0.168 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.209) accounting for .95:
0.168 [[(U43 Other)]]
...
Refining U10 with CSB-INACTIVE
...
Conflict!  There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
```

```
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8173 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[10:36:09] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7450 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N129]]]
        aka [PIN 11 U10] aka [OUT 3 U10A]
        ((T #<ENV 5 0307>))
...
[10:37:13] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N129]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7450 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N280]]]
```

```
        aka [PIN 10 U10] aka [IN ENBT U10A]
        ((T #<ENV 5 0307>))
...
[10:38:16] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N280]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7093 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N236]]]
        aka [PIN 10 RN6] aka [BI 10 RN6A]
        ((T #<ENV 4 0360>))
...
[10:39:19] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N236]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7093 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N117]]]
        aka [PIN 18 U43] aka [IN 1 U43A]
        ((T #<ENV 4 0360>))
...
[10:40:22] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N117]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7093 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N194]]]
        aka [PIN 4 U44] aka [BI 6 U44A]
        ((T #<ENV 4 0360>))
...
[10:41:27] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N194]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7093 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N112]]]
        aka [PIN 21 U43] aka [IN 4 U43A]
        ((T #<ENV 4 0360>))
```

```
...
[10:42:33] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N112]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5676 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N208]]]
        aka [PIN 24 U43] aka [IN 7 U43A]
        ((T #<ENV 3 0320>))
...
[10:43:38] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N208]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5676 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
        aka [PIN 15 U21] aka [BI 5 U21A]
        ((T #<ENV 3 0320>))
...
[10:44:41] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5676 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
        aka [PIN 28 U43] aka [IN 11 U43A]
        ((T #<ENV 3 0320>))
...
[10:45:47] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5512 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N232]]]
        aka [PIN 11 U43] aka [IN WR U43A]
        ((T #<ENV 3 01210>))
...
[10:46:50] Adding observation of T at
```

```
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N232]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4377 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N139]]]
        aka [PIN 13 U21] aka [BI 1 U21A]
        ((T #<ENV 2 0120>))
...
[10:47:58] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N139]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4377 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N131]]]
        aka [PIN 7 U21] aka [BI 0 U21A]
        ((T #<ENV 2 0120>))
...
[10:49:08] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N131]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4377 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N246]]]
        aka [PIN 26 U43] aka [IN 9 U43A]
        ((T #<ENV 2 0120>))
...
[10:50:15] Adding observation of 1 at
 [LL [HOLE 2 N246]]
...
Conflict!  There are 2 diagnoses (entropy 0.923) accounting for .95:
0.660 [[(U21 Other)]]
...
There are 2 diagnoses (entropy 0.923) accounting for .95:
0.660 [[(U21 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.6498 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N159]]]
        aka [PIN 19 U43] aka [IN 2 U43A]
        ((T #<ENV 2 0140>))
...
```

```
[10:51:13] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N159]]]
...
There are 2 diagnoses (entropy 0.923) accounting for .95:
0.660 [[(U21 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.6498 [CHANGING-WRT 0 10000000000 [LL [HOLE 4 N254]]]
        aka [PIN 19 U44] aka [IN S1 U44A]
        ((NIL #<ENV 1 0100>))
...
[10:52:02] Adding observation of 1 at
 [LL [HOLE 4 N254]]
...
Conflict!  There are 1 diagnoses (entropy 0.015) accounting for .95:
0.990 [[(U21 Other)]]
...
There are 1 diagnoses (entropy 0.015) accounting for .95:
0.990 [[(U21 Other)]]
...
Probes      Diagnoses
(Nineteen)  afterwards
      N272  # 1
      N272  ########## 10
      N290  ########## 10
      N129  ########## 10
      N280  ########## 10
      N236  ########## 10
      N117  ########## 10
      N194  ########## 10
      N112  ########## 10
      N208  ########## 10
      N289  ########## 10
       N48  ########## 10
      N232  ########## 10
      N139  ########## 10
      N131  ########## 10
      N246  ## 2
      N159  ## 2
      N254  # 1
T
```

A.8 Audio Decoder Example IV

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[20:15:12] Adding observation of 30 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[20:15:19] Adding observation of 20000.0 at
 [FWW 50000000 '(NIL T) [CROSS (EXPT 2 11) [VOLTAGE [HOLE 1 N272]]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[20:15:24] Adding observation of 20000.0 at
 [FWW 50000000 '(NIL T) [CROSS 0 [DT [VOLTAGE [HOLE 1 N272]]]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U12 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
```

```
...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8291 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[20:19:11] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...

...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7617 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N280]]]
        aka [PIN 15 U11] aka [OUT TC U11A]
        ((T #<ENV 5 0307>))
...
[20:20:14] Adding observation of 1 at
 [LL [HOLE 1 N280]]
...
Conflict!  There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U10 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U11 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U20 WORKING]>)
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...

...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N159]]]
        aka [PIN 8 RN6] aka [BI 8 RN6A]
        ((T #<ENV 4 0360>))
...
[20:22:36] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N159]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...
```

```
...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N194]]]
        aka [PIN 7 RN6] aka [BI 7 RN6A]
        ((T #<ENV 4 0360>))
...
[20:23:31] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N194]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N139]]]
        aka [PIN 23 U43] aka [IN 6 U43A]
        ((T #<ENV 3 0320>))
...
[20:24:37] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N139]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N88]]]
        aka [PIN 25 U43] aka [IN 8 U43A]
        ((T #<ENV 3 0320>))
...
[20:26:23] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N88]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N246]]]
        aka [PIN 5 U21] aka [BI 4 U21A]
        ((T #<ENV 3 0320>))
...
[20:27:21] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N246]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
```

```
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
        aka [PIN 15 U21] aka [BI 5 U21A]
        ((T #<ENV 3 0320>))
...
[20:28:24] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7167 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
        aka [PIN 28 U43] aka [IN 11 U43A]
        ((T #<ENV 3 0320>))
...
[20:29:22] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
...
There are 5 diagnoses (entropy 2.288) accounting for .95:
0.263 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7166 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N223]]]
        aka [PIN 11 U20] aka [IN A U20D]
        ((NIL #<ENV 2 05>))
...
[20:30:20] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N223]]]
...
Conflict!  There are 2 diagnoses (entropy 0.997) accounting for .95:
0.533 [[(U11 Other)]]
...
There are 2 diagnoses (entropy 0.997) accounting for .95:
0.533 [[(U11 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8367 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N101]]]
        aka [PIN 9 U10] aka [IN LOAD U10A]
        ((NIL #<ENV 1 01>))
...
[20:31:12] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N101]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U11 Other)]]
...
Probes     Diagnoses
(Fourteen) afterwards
```

```
      N272  # 1
      N272  ########## 10
      N290  ########## 10
      N280  ##### 5
      N159  ##### 5
      N194  ##### 5
      N139  ##### 5
       N88  ##### 5
      N246  ##### 5
      N289  ##### 5
       N48  ##### 5
      N223  ## 2
      N101  # 1
T
```

A.9 Audio Decoder Example IV with Syndromes

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[09:37:48] Adding observation of 30 at
 [MAX-MIN-WW 100000000 [VOLTAGE [HOLE 1 N272]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[09:37:57] Adding observation of 20000.0 at
 [FWW 50000000 '(NIL T) [CROSS (EXPT 2 11) [VOLTAGE [HOLE 1 N272]]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[09:38:02] Adding observation of 20000.0 at
 [FWW 50000000 '(NIL T) [CROSS 0 [DT [VOLTAGE [HOLE 1 N272]]]]]
...
There are 10 diagnoses (entropy 3.269) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U12 with INACTIVE
...
Conflict!  There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.203) accounting for .95:
0.163 [[(U43 Other)]]
...
Refining U11 with CSB-INACTIVE
...
Conflict!  There are 11 diagnoses (entropy 3.320) accounting for .95:
0.163 [[(U43 Other)]]
...
Conflict!  There are 10 diagnoses (entropy 3.209) accounting for .95:
0.168 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.209) accounting for .95:
0.168 [[(U43 Other)]]
...
Refining U10 with CSB-INACTIVE
...
Conflict!  There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U43 WORKING]>)
```

```
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U21 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U44 WORKING]>)
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8173 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
        aka [PIN 10 U43] aka [IN CS U43A]
        ((T #<ENV 6 01307>))
...
[09:43:12] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N290]]]
...
There are 10 diagnoses (entropy 3.172) accounting for .95:
0.168 [[(U43 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7450 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N280]]]
        aka [PIN 15 U11] aka [OUT TC U11A]
        ((T #<ENV 5 0307>))
...
[09:44:15] Adding observation of 1 at
 [LL [HOLE 1 N280]]
...
Conflict!  There are 7 diagnoses (entropy 2.637) accounting for .95:
0.199 [[(U12 Other)]]
...
Conflict!  There are 6 diagnoses (entropy 2.462) accounting for .95:
0.210 [[(U12 Other)]]
...
Conflict!  There are 5 diagnoses (entropy 2.302) accounting for .95:
0.220 [[(U12 Other)]]
```

```
...
There are 5 diagnoses (entropy 2.302) accounting for .95:
0.220 [[(U12 Other)]]
...
Refining U20 with CSB-INACTIVE
...
Conflict!  There are 5 diagnoses (entropy 2.292) accounting for .95:
0.238 [[(U12 Other)]]
...
There are 5 diagnoses (entropy 2.292) accounting for .95:
0.238 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U12 WORKING]>)
...
There are 5 diagnoses (entropy 2.292) accounting for .95:
0.238 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U10 WORKING]>)
...
There are 5 diagnoses (entropy 2.292) accounting for .95:
0.238 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U11 WORKING]>)
...
There are 5 diagnoses (entropy 2.292) accounting for .95:
0.238 [[(U12 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U20 WORKING]>)
...
There are 5 diagnoses (entropy 2.292) accounting for .95:
0.238 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 1.0000 [LL [HOLE 1 N213]]
        aka [PIN 15 U10] aka [OUT TC U10A]
        ((0 #<ENV 3 04102> #<ENV 3 020102>))
...
[09:48:55] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N213]]]
...
There are 5 diagnoses (entropy 2.293) accounting for .95:
0.239 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 1.0000 [LL [HOLE 2 N56]]
        aka [PIN 12 U21] aka [IN CLOCK U21A]
        ((0 #<ENV 2 04100> #<ENV 2 010100> #<ENV 2 020100>))
...
[09:50:15] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N56]]]
...
```

```
There are 5 diagnoses (entropy 2.293) accounting for .95:
0.239 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7423 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N236]]]
        aka [PIN 14 U44] aka [BI 3 U44A]
        ((T #<ENV 4 0360>))
...
[09:51:24] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N236]]]
...
There are 5 diagnoses (entropy 2.293) accounting for .95:
0.239 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7423 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N117]]]
        aka [PIN 5 U44] aka [BI 4 U44A]
        ((T #<ENV 4 0360>))
...
[09:52:26] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N117]]]
...
There are 5 diagnoses (entropy 2.293) accounting for .95:
0.239 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7423 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N208]]]
        aka [PIN 24 U43] aka [IN 7 U43A]
        ((T #<ENV 3 0320>))
...
[09:53:35] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N208]]]
...
There are 5 diagnoses (entropy 2.293) accounting for .95:
0.239 [[(U12 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.7423 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
        aka [PIN 15 U21] aka [BI 5 U21A]
        ((T #<ENV 3 0320>))
...
[09:54:40] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 1 N289]]]
...
There are 5 diagnoses (entropy 2.293) accounting for .95:
0.239 [[(U12 Other)]]
...
```

```
...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.7423 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
        aka [PIN 28 U43] aka [IN 11 U43A]
        ((T #<ENV 3 0320>))
...
[09:55:42] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N48]]]
...
There are 5 diagnoses (entropy 2.293) accounting for .95:
0.239 [[(U12 Other)]]
...

...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.6803 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N223]]]
        aka [PIN 11 U20] aka [IN A U20D]
        ((NIL #<ENV 2 05>))
...
[09:56:45] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 2 N223]]]
...
Conflict!  There are 2 diagnoses (entropy 0.988) accounting for .95:
0.567 [[(U11 Other)]]
...
There are 2 diagnoses (entropy 0.988) accounting for .95:
0.567 [[(U11 Other)]]
...

...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.8630 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N101]]]
        aka [PIN 9 U10] aka [IN LOAD U10A]
        ((NIL #<ENV 1 01>))
...
[09:57:40] Adding observation of T at
 [CHANGING-WRT 0 10000000000 [LL [HOLE 3 N101]]]
...
There are 1 diagnoses (entropy 0.010) accounting for .95:
0.993 [[(U11 Other)]]
...
Probes     Diagnoses
(Fourteen) afterwards
      N272 # 1
      N272 ########## 10
      N290 ########## 10
      N280 ##### 5
      N213 ##### 5
       N56 ##### 5
      N236 ##### 5
      N117 ##### 5
      N208 ##### 5
      N289 ##### 5
       N48 ##### 5
      N223 ## 2
      N101  # 1
T
```

A.10 Input Encoder Example I

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:19:20] Adding observation of T at
 [POWER [IN POWER S370]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:22:17] Adding observation of 0 at
 [LL [HOLE 2 N83]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:22:27] Adding observation of 1 at
 [LL [HOLE 2 N83]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:23:23] Adding observation of 0 at
 [LL [HOLE 2 N83]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:25:04] Adding observation of NIL at
 [KS [IN PAD U]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:25:06] Adding observation of NIL at
 [KS [IN KBD U]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:25:11] Adding observation of NIL at
 [KT [OUT KEYS C]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:25:12] Adding observation of T at
 [CHANGING-WRT 1000000000 10000000000 [MP [IN MDX U]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
```

```
[16:25:16] Adding observation of T at
 [CHANGING-WRT 1000000000 10000000000 [MP [IN MDY U]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:25:19] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [IN MB U]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[16:25:22] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [OUT MDX C]]]
...
There are 18 diagnoses (entropy 3.932) accounting for .95:
0.136 [[(U25 Other)]]
...
Refining U25 with OPEN
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U25 WORKING]>)
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U33 WORKING]>)
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U34 WORKING]>)
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
[16:31:28] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [OUT MDY C]]]
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
[16:32:39] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [OUT MB C]]]
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.9898 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N178]]]
        aka [PIN 35 U34] aka [BI 20 U34A]
        ((T #<ENV 9 0777>) (NIL #<ENV 14 01777035>))
...
[16:35:43] Adding observation of 1 at
 [LL [HOLE 1 N178]]
...
There are 9 diagnoses (entropy 2.896) accounting for .95:
0.280 [[(U25 Open)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 1.0448 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N257]]]
        aka [PIN 10 RN7] aka [BI 10 RN7A]
        ((T #<ENV 6 077>) (NIL #<ENV 6 02000073>))
...
[16:39:26] Adding observation of 156250.0 at
 [FWW 640000 '(1 0) [LL [HOLE 1 N257]]]
[16:40:20] Adding observation of 156250.0 at
 [FWW 640000 '(0 1) [LL [HOLE 1 N257]]]
...
There are 7 diagnoses (entropy 2.616) accounting for .95:
0.332 [[(U34 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U30 WORKING]>)
...
There are 7 diagnoses (entropy 2.616) accounting for .95:
0.332 [[(U34 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U14 WORKING]>)
...
There are 7 diagnoses (entropy 2.616) accounting for .95:
0.332 [[(U34 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U32 WORKING]>)
...
There are 7 diagnoses (entropy 2.616) accounting for .95:
0.332 [[(U34 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8879 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 3 N162]]]
        aka [PIN 4 U33] aka [IN RESET U33A]
        ((T #<ENV 3 0302>) (NIL #<ENV 6 0437> #<ENV 6 0473>))
...
[16:45:29] Adding observation of 0 at
 [LL [HOLE 3 N162]]
[16:45:41] Adding observation of 1 at
 [LL [HOLE 3 N162]]
...
There are 5 diagnoses (entropy 2.079) accounting for .95:
0.442 [[(U34 Other)]]
...


...
```

```
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4987 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N130]]]
        aka [PIN 6 U32] aka [OUT Y U32C]
        ((T #<ENV 4 035> #<ENV 5 073>) (NIL #<ENV 4 02000031>))
...
[16:47:45] Adding observation of 5000000.0 at
 [FWW 20000 '(0 1) [LL [HOLE 1 N130]]]
...
There are 5 diagnoses (entropy 2.081) accounting for .95:
0.443 [[(U34 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4724 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 2 N243]]]
        aka [PIN 3 U33] aka [IN XTAL2 U33A]
        ((T #<ENV 4 035> #<ENV 5 073>))
...
[16:49:27] Adding observation of 5000000.0 at
 [FWW 20000 '(1 0) [LL [HOLE 2 N243]]]
...
There are 5 diagnoses (entropy 2.081) accounting for .95:
0.443 [[(U34 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4724 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N84]]]
        aka [PIN 10 U32] aka [OUT Y U32E]
        ((T #<ENV 4 035> #<ENV 5 073>))
...
[16:51:01] Adding observation of 5000000.0 at
 [FWW 20000 '(0 1) [LL [HOLE 1 N84]]]
...
There are 5 diagnoses (entropy 2.081) accounting for .95:
0.443 [[(U34 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4724 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 4 N73]]]
        aka [PIN 2 U34] aka [IN XTAL1 U34A]
        ((T #<ENV 4 035> #<ENV 5 073>))
...
[16:52:33] Adding observation of 5000000.0 at
 [FWW 20000 '(1 0) [LL [HOLE 4 N73]]]
...
There are 5 diagnoses (entropy 2.081) accounting for .95:
0.443 [[(U34 Other)]]
...


...
Entropy          Signal; Aliases; Value-Environment Pairs
 0.4724 [CHANGING-WRT 1000000000 10000000000 [CC C5MHZL]]
```

```
        aka [PIN 3 U33] aka [IN XTAL2 U33A]
        ((T #<ENV 4 035> #<ENV 5 073>))
...
[16:54:08] Adding observation of 5000000.0 at
 [FWW 20000 '(NIL T) [CC C5MHZL]]
...
There are 3 diagnoses (entropy 1.321) accounting for .95:
0.624 [[(U34 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.2805 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N45]]]
        aka [PIN 4 U32] aka [OUT Y U32B]
        ((T #<ENV 2 014> #<ENV 3 031>) (NIL #<ENV 2 02000010>))
...
[16:55:41] Adding observation of 1.0e7 at
 [FWW 10000 '(0 1) [LL [HOLE 1 N45]]]
[16:55:52] Adding observation of 1.0e7 at
 [FWW 10000 '(1 0) [LL [HOLE 1 N45]]]
...
There are 3 diagnoses (entropy 1.321) accounting for .95:
0.624 [[(U34 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.2799 [CHANGING-WRT 1000000000 10000000000 [CC C5MHZH]]
        aka [PIN 2 U34] aka [IN XTAL1 U34A]
        ((T #<ENV 1 010>))
...
[16:57:08] Adding observation of 5000000.0 at
 [FWW 20000 '(T NIL) [CC C5MHZH]]
...
There are 2 diagnoses (entropy 0.721) accounting for .95:
0.800 [[(U34 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.2570 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 6 N57]]]
        aka [PIN 9 U26] aka [IN LOAD U26A]
        ((NIL #<ENV 1 02>))
...
[16:58:21] Adding observation of 1 at
 [LL [HOLE 6 N57]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[(U34 Other)]]
...
```

A.11 Input Encoder Example II

```
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:27:37] Adding observation of T at
 [POWER [IN POWER S370]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:30:20] Adding observation of 0 at
 [LL [HOLE 2 N83]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:30:27] Adding observation of 1 at
 [LL [HOLE 2 N83]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:31:02] Adding observation of 0 at
 [LL [HOLE 2 N83]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:33:20] Adding observation of NIL at
 [KS [IN PAD U]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:33:25] Adding observation of NIL at
 [KS [IN KBD U]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:33:30] Adding observation of NIL at
 [KT [OUT KEYS C]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:33:31] Adding observation of T at
 [CHANGING-WRT 1000000000 10000000000 [MP [IN MDX U]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
```

```
[10:33:34] Adding observation of T at
 [CHANGING-WRT 1000000000 10000000000 [MP [IN MDY U]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:33:38] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [IN MB U]]]
...
There are 1 diagnoses (entropy 0.000) accounting for .95:
1.000 [[]]
...
[10:33:42] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [OUT MDX C]]]
...
There are 18 diagnoses (entropy 3.932) accounting for .95:
0.136 [[(U25 Other)]]
...
Refining U25 with OPEN
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U25 WORKING]>)
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U33 WORKING]>)
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U34 WORKING]>)
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
[10:40:29] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [OUT MDY C]]]
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...
[10:41:29] Adding observation of NIL at
 [CHANGING-WRT 1000000000 10000000000 [MP [OUT MB C]]]
...
There are 18 diagnoses (entropy 3.931) accounting for .95:
0.135 [[(U25 Open)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.9898 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 2 N178]]]
        aka [PIN 1 U33] aka [IN TO U33A]
        ((T #<ENV 9 0777>) (NIL #<ENV 14 01777035>))
...
[10:44:28] Adding observation of T at
 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 2 N178]]]
...
There are 13 diagnoses (entropy 3.465) accounting for .95:
0.179 [[(U25 Open)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.8188 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N257]]]
        aka [PIN 10 RN7] aka [BI 10 RN7A]
        ((T #<ENV 6 077>) (NIL #<ENV 6 02000073>))
...
[10:46:41] Adding observation of 156250.0 at
 [FWW 640000 '(1 0) [LL [HOLE 1 N257]]]
[10:47:11] Adding observation of 156250.0 at
 [FWW 640000 '(0 1) [LL [HOLE 1 N257]]]
...
There are 12 diagnoses (entropy 3.404) accounting for .95:
0.165 [[(U33 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U16 WORKING]>)
...
There are 12 diagnoses (entropy 3.404) accounting for .95:
0.165 [[(U33 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.3894 [WRT 1000000002 2052428803 [LL [HOLE 1 N81]]]
        aka [PIN 5 U22] aka [IN B U22B]
        ((1 #<ENV 8 0100373> #<ENV 8 0100337>))
...
[10:49:47] Adding observation of 1 at
 [LL [HOLE 1 N81]]
...
There are 12 diagnoses (entropy 3.404) accounting for .95:
0.165 [[(U33 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.3894 [WRT 1000000002 2052428803 [LL [HOLE 4 N137]]]
        aka [PIN 10 U34] aka [IN WR U34A]
        ((1 #<ENV 8 0100373> #<ENV 8 0100337>))
...
[10:51:39] Adding observation of 1 at
 [LL [HOLE 4 N137]]
...
There are 12 diagnoses (entropy 3.404) accounting for .95:
0.165 [[(U33 Other)]]
...
```

```
...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.3894 [WRT 1000000002 2052428803 [LL [HOLE 2 N11]]]
        aka [PIN 22 U16] aka [IN OE U16A]
        ((1 #<ENV 8 0100373> #<ENV 8 0100337>))
...
[10:53:28] Adding observation of 1 at
 [LL [HOLE 2 N11]]
...
There are 4 diagnoses (entropy 1.770) accounting for .95:
0.488 [[(U33 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U30 WORKING]>)
...
There are 4 diagnoses (entropy 1.770) accounting for .95:
0.488 [[(U33 Other)]]
...
Decomposing (#<ASSUMPTION +INF [STATUS-OF U32 WORKING]>)
...
There are 4 diagnoses (entropy 1.770) accounting for .95:
0.488 [[(U33 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.5257 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 1 N130]]]
        aka [PIN 6 U32] aka [OUT Y U32C]
        ((T #<ENV 4 035> #<ENV 5 073>) (NIL #<ENV 4 02000031>))
...
[10:56:36] Adding observation of 5000000.0 at
 [FWW 20000 '(0 1) [LL [HOLE 1 N130]]]
...
There are 4 diagnoses (entropy 1.771) accounting for .95:
0.490 [[(U33 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.4973 [CHANGING-WRT 1000000000 10000000000 [LL [HOLE 4 N73]]]
        aka [PIN 2 U34] aka [IN XTAL1 U34A]
        ((T #<ENV 4 035> #<ENV 5 073>))
...
[10:59:05] Adding observation of 5000000.0 at
 [FWW 20000 '(1 0) [LL [HOLE 4 N73]]]
...
There are 4 diagnoses (entropy 1.771) accounting for .95:
0.490 [[(U33 Other)]]
...


...
Entropy         Signal; Aliases; Value-Environment Pairs
 0.4973 [CHANGING-WRT 1000000000 10000000000 [CC C5MHZL]]
        aka [PIN 3 U33] aka [IN XTAL2 U33A]
        ((T #<ENV 4 035> #<ENV 5 073>))
```

```
...
[11:00:39] Adding observation of 5000000.0 at
 [FWW 20000 '(NIL T) [CC C5MHZL]]
...
There are 1 diagnoses (entropy 0.048) accounting for .95:
0.966 [[(U33 Other)]]
...
```

Appendix B

Abstractions and Behaviors

The following definitions are discussed in Chapter 5 and collected here alphabetically. Keep in mind that their purpose is mental hygiene, not execution. The procedural style of definition has advantages of expressiveness that outweigh its disadvantages. The expressiveness advantage is that the language is simple enough that it is possible to introduce compound definitions through composition of existing definitions and surface transformations of the results; the **rh** behavior (Appendix C) is an example of this technique. The disadvantages that such transformations would be difficult to automate and intractable in general is a long-term concern, but not an overriding one. Likewise, the inefficiency of the procedures defined in some cases is not a concern, since the troubleshooting program does not use these procedures directly. Finally, there is no compelling reason that a more declarative representation could not have been used – but the same tractability problems would still arise: using a temporal logic as in [Moszkowski82], for example, would not solve the problem of transformations being intractable.

```
accumulated-bits ==
  (lambda (S V D)
    (lambda (time)
      (if (S time) 0
          (let ((previous
                  ((accumulated-bits S V D) (- time δ))))
            (if (V time)
                (+ (if (eql (D time) 1) 1 0)
                   previous)
                previous))))))
```

```
brightness ==
  (lambda (R C1 C2 Kbd Kpd M)
    (lambda (time)
     (let ((the-state ((c-state R C1 C2 Kbd Kpd M) time)))
       (if (eql 'init the-state) 128
           (max 0
            (min 255
             (+ (if (eql 'local the-state)
                    (/ (- ((duration
                             (key-is-pressed 'B Kbd)) time)
                          ((duration
                             (key-is-pressed 'D Kbd)) time))
                       3msec) 0)
                 ((brightness R C1 C2 Kbd Kpd M)
                  (- time
                     ((duration
                        (c-state R C1 C2 Kbd Kpd M))
                      time))))))))))


changing-wrt ==
 (lambda (lb ub S)
   (lambda (time)
     (and (= time ub)
          (> ((count-ww (- ub lb) (change S)) time) 0))))


count-ww ==
  (lambda (n S)
    (lambda (time)
      (if (<= n 0) 0
          (+ (if (S time) 1 0)
             ((count-ww (- n δ) S) (- time δ))))))


cross ==
  (lambda (v S)
    (lambda (time)
      (let ((s0 (S (- time δ δ)))
            (s2 (S time)))
        (or (< s0 v s2) (< s2 v s0)))))


cycles-ww ==
  (lambda (n 1 S)
    (count-ww n (sequence 1 S)))
```

```
dt ==
  (lambda (S)
    (lambda (time)
      (let ((s0 (S (- time δ)))
            (s1 (S time)))
        (/ (- s1 s0) δ))))


duration ==
  (lambda (S)
    (lambda (time)
      (if ((change S) time) δ
          (+ δ ((duration S) (- time δ))))))


fall ==
  (lambda (C)
    (lambda (time)
      (and (= 0 (C time))
           (= 1 (C (- time δ))))))


fww ==
  (lambda (n l S)
    (lambda (time)
      (/ (cycles-ww n l S) n)))


event ==
  (lambda (from to S)
    (lambda (time)
      (and (equal (S time) to)
           (not (equal (S (- time δ)) to))
           (or (eql from :any)
               (equal from (S (- time δ)))))))


gray-event ==
  (lambda (S0 S1)
    (lambda (time)
      (or ((change S0) time) ((change S1) time))))


kbd-events ==
  (keyboard-events kbd-state)
```

```
kbd-state ==
(samp (fall kbd-reset)
  (accumulated-bits
    (fall kbd-reset)
    (rise kbd-clk) kbd-data))


keyboard-events ==
  (lambda (S)
    (lambda (time)
      (if ((stay S) time) nil
          (let ((previous (S (- time δ)))
                (current (S time)))
            (list
              (pos->key (log (logxor previous current) 2))
              (if (< previous current) 'up 'down))))))


key-is-pressed ==
  (lambda (key Kbd)
    (lambda (time)
      (if (eql (list 'up key) (Kbd time)) t
          (if (eql (list 'down key) (Kbd time)) nil
              ((key-is-pressed key Kbd) (- time δ))))))


mouse-dx ==
 (tsign
   (count-ww 1sec
     (gray-event mouse-left mouse-right)))


register ==
  (lambda (C D)
    (lambda (time)
      (if ((fall C) time) (D time)
          ((register C D) (- time δ)))))


samp == sample-and-hold ==
  (lambda (V S)
    (lambda (time)
      (if (V time) (S time)
          ((samp V S) (- time δ)))))
```

```
sequence ==
  (lambda (l S)
    (lambda (time)
      (or (null l)
          (if ((stay S) time)
              ((sequence l S) (- time δ))
              (and (eql (car (last l)) (S (- time δ)))
                   ((sequence (butlast l) S)
                    (- time δ)))))))

sign ==
  (lambda (x) (if (< x 0) '- (if (> x 0) '+ 0)))

synchronous-delay ==  syn-del ==
  (lambda (n V S)
    (lambda (time)
      (if (V time)
          (if (= n 0)
              (S time)
              ((synchronous-delay (- n 1) V S) (- time δ)))
          ((synchronous-delay n V S) (- time δ)))))

syn-register ==
  (lambda (V S) (synchronous-delay 1 V S))

toggle ==
  (lambda (S)
    (lambda (time)
      (if ((fall S) time)
          (invert ((toggle S) (- time δ)))
          ((toggle S) (- time δ)))))

tsign ==
  (lambda (S)
    (lambda (time)
      (sign (S time))))

two-phase-clock ==
 (lambda (phi1 phi2)
   (sequence '((0 0) (1 0) (0 0) (0 1))
     (lambda (time) (list (phi1 time) (phi2 time)))))
```

Appendix C

Reset Hold Counter Behavior

Section 5.8.1 alluded to the fact that the temporally abstract behavior for the Reset Hold Counter component could be derived from the behaviors of its subcomponents. The actual transformations are given here.

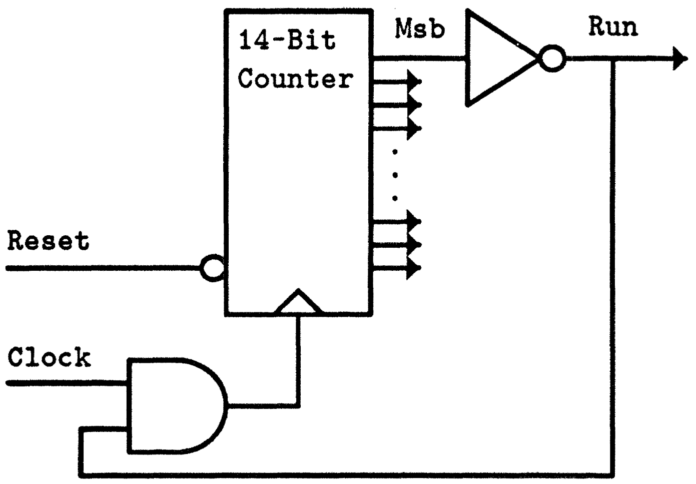
Consider the behaviors of the three components of the Reset Hold Counter (Figure C.1). The behavior of the inverter is **tinvert**, the behavior of the AND gate is **tand**, and the k-bit counter's behavior is represented by **counter** (**nthbit** is an auxiliary function, not a behavior).

```
tand ==
  (lambda (X Y)
    (lambda (time)
      (if (and (eql (X time) 1) (eql (Y time) 1)) 1 0)))


counter ==
  (lambda (k R C)
    (lambda (time)
      (if (eql 0 (R time)) 0
          (mod
            (+ (if ((fall C) time) 1 0)
               ((counter k R C) (- time δ)))
            (expt 2 k)))))


nthbit ==
  (lambda (i n) (load-byte n i 1))
```

Figure C.1: Reset Hold Counter



The behavior of the connected group of components is represented by **rh-state**, which returns a signal representing the state of the Reset Hold Counter. For the most part it is simply a composition of the **counter**, **tand**, and **tinvert** behaviors that reflects the circuit structure. The signal argument to **tinvert** is a delayed version of the most significant bit of state and prevents **rh-state** from being circularly defined; the delay could have been introduced anywhere in the loop. The behavior **rh** is then the behavior of the entire aggregate structure; it is simply the most significant bit of the state.

```
rh ==
 (lambda (R C)
   (lambda (time)
     (nthbit 13 ((rhstate R C) time))))


rh-state ==
  (lambda (R C)
    (counter 14 R
      (tand C
        (tinvert
          (lambda (time)
            (nthbit 13
              ((rh-state R C) (- time delta)))))))
```

Defining the behavior rh and its underlying behavior rh-state does not simplify anything, it merely composes the several behaviors into one.

```
rh-state ==
  (lambda (R C)
    (counter 14 R
      (tand C
        (tinvert
          (lambda (time)
            (nthbit 13
              ((rh-state R C) (- time delta))))))))
```

```
rh ==
 (lambda (R C)
   (lambda (time)
     (nthbit 13 ((rhstate R C) time))))
```

The following transformations simplify rh-state's definition so that it takes on values from 0 to 2^{13} instead of 0 to 2^{14}:
The use of counter is removed by substitution:

```
 (lambda (R C)
   ((lambda (k R C)
      (lambda (time)
        (if (eql 0 (R time)) 0
            (mod
              (+ (if ((fall C) time) 1 0)
                 ((counter k R C) (- time delta)))
              (expt 2 k)))))
     14 R
      (tand C
        (tinvert
          (lambda (time)
            (nthbit 13
              ((rh-state R C) (- time delta))))))))
```

Substitution for k, R, and C promotes the (eql 0 (R time)) condition:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        ((lambda (CC)
           (mod
             (+ (if ((fall CC) time) 1 0)
                ((counter 14 R CC) (- time delta)))
             (expt 2 14)))
         (tand
           C (tinvert
               (lambda (time)
                 (nthbit
                   13 ((rh-state R C)
                       (- time delta))))))))))
```

The term (counter 14 R CC) is equivalent to (rh-state R C) and can be substituted:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        ((lambda (CC)
           (mod
             (+ (if ((fall CC) time) 1 0)
                ((rh-state R C) (- time delta)))
             (expt 2 14)))
         (tand
           C (tinvert
               (lambda (time)
                 (nthbit
                   13 ((rh-state R C)
                       (- time delta))))))))))
```

With only one reference to CC remaining, it can be substituted for:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        (mod
          (+ (if ((fall
                    (tand
                      C (tinvert
                          (lambda (time)
                            (nthbit
                              13 ((rh-state R C)
                                   (- time delta)))))))
                 time) 1 0)
             ((rh-state R C) (- time delta)))
          (expt 2 14)))))
```

We can now case split on whether the term (nthbit 13 ...) is 1 or 0:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        (if (eql 0 (nthbit
                     13 ((rh-state R C) (- time delta))))
            (mod
              (+ (if ((fall C)
                      time) 1 0)
                 ((rh-state R C) (- time delta)))
              (expt 2 14))
            (mod
              (+ (if ((fall (lambda (time) 0))
                      time) 1 0)
                 ((rh-state R C) (- time delta)))
              (expt 2 14))))))
```

Simplifying the else-part of the resulting condition yields:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        (if (eql 0 (nthbit
                      13 ((rh-state R C) (- time delta))))
            (mod
              (+ (if ((fall C)
                      time) 1 0)
                 ((rh-state R C) (- time delta)))
              (expt 2 14))
            ((rh-state R C) (- time delta))))))
```

The condition `(eql 0 (nthbit 13 x))` can be expressed in an alternative way as `(< x (expt 2 13))`:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        (if (< (expt 2 13)
               ((rh-state R C) (- time delta)))
            (mod
              (+ (if ((fall C)
                      time) 1 0)
                 ((rh-state R C) (- time delta)))
              (expt 2 14))
            ((rh-state R C) (- time delta))))))
```

This allows us to drop the `mod` term from the if-part:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        (if (< (expt 2 13)
               ((rh-state R C) (- time delta)))
            (+ (if ((fall C)
                    time) 1 0)
               ((rh-state R C) (- time delta)))
            ((rh-state R C) (- time delta))))))
```

Finally, the conditional can be formulated as a `min` expression:

```
(lambda (R C)
  (lambda (time)
    (if (eql 0 (R time)) 0
        (min (expt 2 13)
             (+ (if ((fall C) time) 1 0)
                ((rh-state R C) (- time delta)))))))
```

The following schema says that moment-by-moment conditional counting of `?Y` can be replaced with "jumps" of duration `?n`, when `?X` is periodic and `?F` is monotonic:

```
?SELF ==
 (lambda (?X ?Y)
   (lambda (time)
     (if (?X time) 0
         (?F (+ (if (?Y time) 1 0)
                ((?SELF ?X ?Y) (- time delta)))))))

                          ==>

?SELF ==
  (lambda (?X ?Y)
    (lambda (time)
      (if (X? time) 0
          (let ((N (count-ww ?n ?Y)))
            (?F (if (< n ((duration ?X) time))
                    (+ (N time) ((?SELF ?X ?Y) (- time ?n)))
                    (* (N time)
                       (/ ((duration ?X) time) ?n))))))))
```

When this transformation is applied to a rewritten definition of **rh-state** called **new-rh-state**, the following results:

```
rh-state ==
  (lambda (R C)
    ((lambda (LR FC)
       (lambda (time)
         (if (LR time) 0
             ((lambda (x) (min (expt 2 13) x))
              (+ (if (FC time) 1 0)
                 ((new-rh-state LR FC) (- time delta)))))))
     (lambda (time) (eql 0 (R time)))
     (fall C)))
```

Which becomes, with `?n` still unbound:

```
new-rh-state ==
  (lambda (R C)
    ((lambda (LR FC)
       (lambda (time)
         (if (LR time) 0
             (let ((NFC (count-ww ?n FC)))
               ((lambda (x) (min (expt 2 13) x))
                (if (< n ((duration LR) time))
                    (+ (NFC time)
                       ((rh-state LR NFC) (- time n)))
                    (* (NFC time)
                       (/ ((duration LR) time) n))))))))
     (lambda (time) (eql 0 (R time)))
     (fall C)))
```

We can make further use of the assumption that `C` is periodic by using the *frequency* temporal abstraction to describe `C`, and expressing `rh-state` in terms of that abstraction. The transformations required to do the latter are as follows; first the `FC` argument is substituted for the original term `(fall C)`:

```
rh-state ==
  (lambda (R C)
    ((lambda (LR)
       (lambda (time)
         (if (LR time) 0
             (let ((NFC (count-ww ?n (fall C))))
               ((lambda (x) (min (expt 2 13) x))
                (if (< n ((duration LR) time))
                    (+ (NFC time)
                       ((rh-state LR NFC) (- time n)))
                    (* (NFC time)
                       (/ ((duration LR) time) n))))))))
     (lambda (time) (eql 0 (R time)))))
```

The term `(cycles-ww n '(0 1) C)` is then substituted for the equivalent term `(count-ww n (fall C))`:

```
rh-state ==
  (lambda (R C)
    ((lambda (LR)
       (lambda (time)
         (if (LR time) 0
             (let ((NFC (cycles-ww ?n '(0 1) C)))
               ((lambda (x) (min (expt 2 13) x))
                (if (< n ((duration LR) time))
                    (+ (NFC time)
                       ((rh-state LR NFC) (- time n)))
                    (* (NFC time)
                       (/ ((duration LR) time) n))))))))
     (lambda (time) (eql 0 (R time)))))
```

The `cycles` abstraction can be reformulated in terms of `fww` as follows:

```
rh-state ==
  (lambda (R C)
    ((lambda (LR)
       (lambda (time)
         (if (LR time) 0
             (let ((NFC (lambda (time)
                          (* n ((fww ?n '(0 1) C) time)))))
               ((lambda (x) (min (expt 2 13) x))
                (if (< n ((duration LR) time))
                    (+ (NFC time)
                       ((rh-state LR NFC) (- time n)))
                    (* (NFC time)
                       (/ ((duration LR) time) n))))))))
     (lambda (time) (eql 0 (R time)))))
```

Now `NFC` can be substituted into the body:

```
rh-state ==
  (lambda (R C)
    ((lambda (LR)
       (lambda (time)
         (if (LR time) 0
             ((lambda (x) (min (expt 2 13) x))
              (if (< n ((duration LR) time))
                  (+ (* n ((fww ?n '(0 1) C) time))
                     ((rh-state LR NFC) (- time n)))
                  (* (* n ((fww ?n '(0 1) C) time))
                     (/ ((duration LR) time) n)))))))
     (lambda (time) (eql 0 (R time)))))
```

and the common subexpression promoted, with a simplification in the else-part of the `if`:

```
rh-state ==
  (lambda (R C)
    ((lambda (LR)
       (lambda (time)
         (if (LR time) 0
             ((lambda (x) (min (expt 2 13) x))
              (let ((f ((fww ?n '(0 1) C) time)))
                (if (< n ((duration LR) time))
                    (+ (* n f)
                       ((rh-state LR NFC) (- time n)))
                    (* f ((duration LR) time))))))))
     (lambda (time) (eql 0 (R time)))))
```

Since `R` only takes on the values `0` and `1`, `LR` can be removed:

```
rh-state ==
  (lambda (R C)
    (lambda (time)
      (if (eql 0 (R time)) 0
          ((lambda (x) (min (expt 2 13) x))
           (let ((f ((fww ?n '(0 1) C) time)))
             (if (< n ((duration R) time))
                 (+ (* n f) ((rh-state R NFC) (- time n)))
                 (* f ((duration R) time))))))))
```

Finally, the assumption that `C` is periodic can be used. If `C` is periodic, then `f` must be a constant and `?n` is infinite. These substitutions yield:

```
rh-state ==
  (lambda (R C)
    (lambda (time)
      (if (eql 0 (R time)) 0
          ((lambda (x) (min (expt 2 13) x))
           (if (< infinity ((duration R) time))
               (+ (* infinity f)
                  ((rh-state R NFC) (- time infinity)))
               (* i ((duration R) time)))))))
```

A final transformation removes the `if` statement since its condition is always `nil`, and substitutes for `x` (the latter could have been done earlier):

```
rh-state ==
  (lambda (R C)
    (lambda (time)
      (if (eql 0 (R time)) 0
          (min (expt 2 13) (* f ((duration R) time))))))
```

Appendix D

Audio Counter Behavior

Section 5.8.2 alluded to the derivation of the temporally abstract behavior of the Audio Counter; this derivation is presented here.

While the Reset Hold Counter's **Reset** input starts the counter back at 0 whenever asserted, in the Audio Counter only the first 1-to-0 transition of the **Start** signal matters. Eighteen clock cycles must pass before the "start" state can be reached again: while counting, it is insensitive to the **Start** signal. One consequence is that while the transformation from the directly composed behavior of the Reset Hold Counter to a simplified behavior was tedious but straightforward, the simplified behavior of the Audio Counter is not much of an improvement over the composed behavior, and seems to be derivable only by expanding the behavior to an eighteen-way case split and then collapsing it.

The four-bit counters are both wired to load "14" when the Load signal goes low:

```
k-bit-counter-with-synchronous-clear-state ==
  (lambda (k D L P T C)
    (lambda (time)
      (let ((previous ((self L P T C) (- time δ))))
        (if ((rise C) time)
            (if (eql 0 (L time)) (D time)
                (mod (expt 2 k)
                     (+ (if (and (eql 1 (P time))
                                 (eql 1 (T time)))
                            1 0)
                        previous)))
            previous))))
```

```
four-bit-counter-with-synchronous-clear-state ==
  (lambda (L P T C)
    (k-bit-counter-with-synchronous-clear-state
      4 (lambda (time) 14) L P T C))
```

The composition and simplification of the behaviors of those two counters results in the following similar behavior:

```
eight-bit-counter-with-synchronous-clear-state ==
  (lambda (L P T C)
    (k-bit-counter-with-synchronous-clear-state
      8 (lambda (time) (- 64 18)) L P T C))
```

Including the feedback signal Msb results in the following composed definition:

```
eighteen-counter ==
  (lambda (S C)
    (nthbit 8 (rising-edge-eighteen-counter-state S C)))


rising-edge-eighteen-counter-state ==
(lambda (S C)
  (lambda (time)
    (let ((L (lambda (time)
               (nthbit
                 8 ((rising-edge-eighteen-counter-state S C)
                    (- time δ))))))
      (eight-bit-counter-with-synchronous-clear-state
        (tinvert (tnor S L))
        L (lambda (time) 1) C))))
```

Finally, after many transformations the following simplified definition results:

```
rising-edge-eighteen-counter-state ==
  (lambda (S C)
    (lambda (time)
      (let ((previous
              ((rising-edge-eighteen-counter-state S C)
               (- time δ))))
        (if ((rise C) time)
            (if (eql 0 (S time)) (- 64 18)
                (if (eql previous 0) 0
                    (mod (+ 1 previous) 64)))
            previous))))
```

Some temporal abstractions that applied to the Reset Hold Counter can be applied to this simplified behavior; however, the assumptions on which they depend are violated by the normal usage of the circuit and so the resulting temporally abstract behaviors have little predictive force. For example, while the signal `Msb` is a constant `1`, the Audio Counter forms a frequency divider with respect to the `Clock` input; however, the clocks come in bursts of 18 and normally the `Start` line goes low at least once per burst — the "frequencies" are thus not constant, but rather are defined over so few cycles as to be useless. For another example, the "counting" behavior of the Audio Counter can be captured by `(* (duration S) (fww n '(1 0) C))` only during the bursts of 18 clock cycles and hence is similarly useless.

The behavior as shown above is not event-preserving with respect to `S`: any number of events could happen while `C` had no rising edges, and in that case `Msb` would not change. However, the *Sampling* abstraction, when applied to the `Start`, `Load`, and `Msb` signals with respect to the temporally abstract signal `(rise Clock)`, yields the following slightly modified behavior that is event preserving:

```
rising-edge-eighteen-counter-state ==
  (lambda (S C)
    ((lambda (SS)
       (lambda (time)
         (let ((previous
                 ((rising-edge-eighteen-counter-state S C)
                  (- time δ))))
           (if ((rise C) time)
               (if (eql 0 (SS time)) (- 64 18)
                   (if (eql previous 0) 0
                       (mod (+ 1 previous) 64)))
                   previous)))))
     (samp (rise C) S)))
```

`(lambda (SS) ...)` is event-preserving, to the extent that n falling edges on `(samp (rise C) S)` will result in somewhere between $\lfloor\frac{n}{18}\rfloor$ and n falling edges on `Msb`. This is because `(samp (rise C) S)` can only change at the same moments that `C` rises. Thus the number of falls on `Msb` (measured with respect to rising edges of `Clock`) is bounded as follows:

```
  ((count-ww
     n (fall (samp (rise Clock) Start))) time)  ≥
    ((count-ww
       n (fall (samp (rise Clock) Msb))) time)  ≥
(floor
  ((count-ww
     n (fall (samp (rise Clock) Start))) time)
  18)
```

Appendix E

The Switch Level Model

The lowest level of circuit description in BASIL is a switch level model. The primitive elements of the model are pins, etches, resistors, switches, and voltage-controlled switches (that is, transistors). The model uses voltages in the set {`0`,`1`} and currents in the set {`-`,`0`,`+`} with `0` meaning "negligible." This models the steady-state digital behavior of simple analog elements. The digital current model is needed because circuit boards contain physical switches, jumpers, and resistors, whose behavior cannot be modeled adequately by a gate-level digital model.

E.1 Pins and other Connections

Behaviorally, the simplest elements are connections, which have ports at two ends. Working connections transmit certain signals unchanged from one port to the other. The signals thus transmitted are called *ordinary* signals; `voltage` is the most primitive such signal, and most abstractions of it including `logic-level` are ordinary signals as well. Using a demon facility instead of rules, each signal that appears at one end of a connection can result in that signal getting equated (via `tsame`) to the corresponding signal at the other end. For example, as long as the connection `c` is working, the logic-level signals at either end carry the same value:

```
[conn c (out 0 a) (in 0 b)]
[status-of c working]
Signal (ll (out 0 a)) exists
(ll (out 0 a)) is an ordinary signal
→
[tsame -∞ +∞ (ll (out 0 a)) (ll (in 0 b))]
```

Pins are a kind of connection, and they transmit ordinary signals in this fashion.

Just as there are logic-level signals denoted (ll X) and representing a function from time to {0,1}, there are *qualitative-current-into* signals denoted (qci X). Qualitative currents range over {-,0,+}, and have the arithmetic operations *qplus* and *qminus* with their usual meanings. Pins obey a qualitative version of Kirchhoff's current law (KCL); that is, the sum of the currents into the pin must be 0:

```
  If  [conn (pin ?n ?chip) ?source ?sink]
 and  [status-of (pin ?n ?chip) working]
 and  [thru ?l ?u (qci ?source) ?i]
Then  [thru ?l ?u (qci ?sink) (qminus ?i)]

  If  [conn (pin ?n ?chip) ?source ?sink]
 and  [status-of (pin ?n ?chip) working]
 and  [thru ?l ?u (qci ?sink) ?i]
Then  [thru ?l ?u (qci ?source) (qminus ?i)]
```

Etches obey similar rules as pins, although they can have any number of ports denoted (hole 1 ...), (hole 2 ...), and so forth. The number of ports on an etch is referred to as its "arity." To transmit ordinary signals, n rules could be written for each arity, one that says that the value at hole 1 is the same as at hole 2, the value at hole 2 is the same as at hole 3, and so forth. Since some etches have several dozen ports, this is impractical and inefficient. Instead, BASIL defines for each etch a distinguished port not corresponding to any physical boundary, which TINT connects to each hole by a binary connection. For example, suppose etch n119 has arity 3. In addition to its three ports (hole 1 n119), (hole 2 n119), and (hole 3 n119), it has a port LL119 to which all three are connected.

Etches also have qualitative KCL rules, and the rules for an etch with 3 holes are shown below; n-ary etches require n rules of this form:

```
  If  [isa ?e etch]
 and  [status-of ?e working]
 and  [thru ?l2 ?u2 (qci (hole 2 ?e)) ?i2]
 and  [thru ?l3 ?u3 (qci (hole 3 ?e)) ?i3]
 and  (overlap (?l2 ?u2) (?l3 ?u3))
Then  [thru (max ?l2 ?l3) (min ?u2 ?u3)
            (qci (hole 1 ?e)) (qminus (qplus ?i2 ?i3))]

  If  [isa ?e etch]
 and  [status-of ?e working]
 and  [thru ?l1 ?u1 (qci (hole 1 ?e)) ?i1]
 and  [thru ?l3 ?u3 (qci (hole 3 ?e)) ?i3]
 and  (overlap (?l1 ?u2) (?l1 ?u3))
Then  [thru (max ?l1 ?l3) (min ?u1 ?u3)
            (qci (hole 2 ?e)) (qminus (qplus ?i1 ?i3))]

  If  [isa ?e etch]
 and  [status-of ?e working]
 and  [thru ?l1 ?u1 (qci (hole 1 ?e)) ?i1]
 and  [thru ?l2 ?u2 (qci (hole 2 ?e)) ?i2]
 and  (overlap (?l1 ?u2) (?l1 ?u2))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
            (qci (hole 3 ?e)) (qminus (qplus ?i1 ?i2))]
```

E.2 Resistors

Resistors have (i) positive resistance, (ii) two ports `(bi 1 ...)` and `(bi 2 ...)`, and (iii) rules enforcing KCL. The mode of a resistor is normal if the resistor is working. While in normal mode it obeys a qualitative version of Ohm's law embodied as two rules. First, the current into a resistor has the same sign as the voltage drop across it:

```
  If  [isa ?r resistor]
 and  [thru ?l1 ?u1 (mode ?r) normal]
 and  [thru ?l2 ?u2 (l1 (bi 1 ?r)) ?v1]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
 and  [thru ?l3 ?u3 (l1 (bi 2 ?r)) ?v2]
 and  (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
Then  [thru (max ?l1 ?l2 ?l3) (min ?u1 ?u2 ?u3)
            (qci (bi 1 ?r)) (sign (- ?v1 ?v2))]
```

Second, if there is no current flowing into a resistor then there is no voltage drop across it; that is, the logic-levels at both ends are the same:

```
  If  [isa ?r resistor]
 and  [thru ?l1 ?u1 (mode ?r) normal]
 and  [thru ?l2 ?u2 (qci (bi ?n ?r)) 0]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
Then  [tsame (max ?l1 ?l2) (min ?u1 ?u2)
             (l1 (bi 1 ?r)) (l1 (bi 2 ?r))]
```

The second rule could be generalized; nonzero current flowing into a resistor implies that there must be a voltage drop across it. In the implementation, however, every resistor in the Console Controller Board has one end connected to Vdd, so that the two rules above were sufficient and the more general version was never needed.

E.3 Switches

Switches appear on circuit boards in various guises; as jumpers, buttons, or as literal switches whose position the user sets. An ordinary switch has two ports `(bi 1 ...)` and `(bi 2 ...)`, and two modes, **open** and **shut**. In these two modes it either has infinite or negligible resistance, respectively. There are three rules describing the behavior of switches. First, if a switch is open then all the currents into it are 0:

```
  If  [isa ?s switch]
 and  [thru ?l ?u (mode ?s) open]
Then  [thru ?l ?u (qci (bi 1 ?s)) 0]
 and  [thru ?l ?u (qci (bi 2 ?s)) 0]
```

Second, if a switch is shut then there is no voltage drop across it; the logic-levels at its ports are the same:

```
  If  [isa ?s switch]
 and  [thru ?l ?u (mode ?s) shut]
Then  [tsame ?l ?u (ll (bi 1 ?s)) (ll (bi 2 ?s))]
```

Third, if a switch is shut it obeys KCL; that is, if the current into one port is known then the current into the other is its negative:

```
  If  [isa ?s switch]
 and  [thru ?l1 ?u1 (mode ?s) normal]
 and  [thru ?l2 ?u2 (qci (bi ?n ?s)) ?i]
 and  (overlap (?l1 ?u1) (?l2 ?u2))
Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
             (qci (bi (- 3 ?n) ?s)) (qminus ?i)]
```

A typical circuit structure encountered on digital boards is a combination of a switch to ground and a resistor to a constant high voltage (Figure E.1). When the switch is open, the logic-level of node N goes to 1; when shut it is 0 and current flows out of the resistor through the switch to ground.

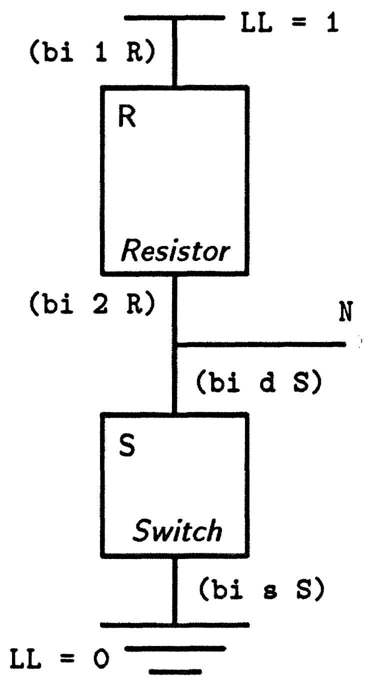
Since the resistor and switch typically belong to different field-replaceable units, it is important in a troubleshooting context for TINT to be able to model at this level of detail.

This level of detail would also be useful for proper handling of failures such as solder bridges and other kinds of "shorts." Although handling of shorts is not implemented, some of the necessary behavior models are in fact included in TINT and so are presented here for completeness.

Transistors are modeled as voltage-controlled switches. Their rules are similar to those for switches, except that the logic-level at their g port determines whether they are open or shut:

```
  If  [isa ?x transistor]
 and  [status-of ?x working]
 and  [thru ?l ?u (ll (in g ?x)) ?v]
Then  [thru ?l ?u (mode ?x)
             (if (eql ?v 0) 'open 'shut)]
```

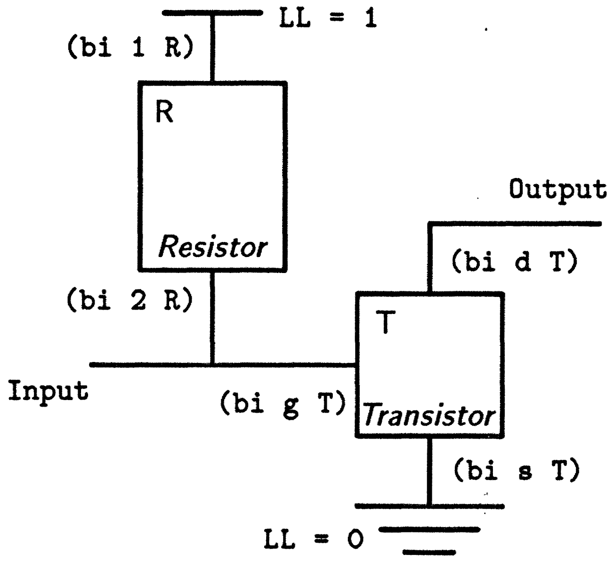
Figure E.1: Typical Switch-Resistor Combination



The resistor and transistor models can be composed to form behavior models of ordinary digital components such as logic gates. The advantage of this level of detail is that the effects of faults that cause shorts between signals (other than power signals) can be correctly modeled. Using the standard digital model, for example, the logic-level output of a working TTL inverter must be 1 if the input is 0, and if it is not, then the inverter *must* be broken. By taking currents into account, the more accurate prediction can be made that if the input logic-level is 0, the output current is 0. Hence, if the output logic-level is 0 instead of 1, it is *not* a necessary logical consequence that the inverter is broken; something else could be pulling the output node down.

Using the switch model the behavior of a TTL inverter can be summarized as follows: if the input current is 0 the output voltage will be 0; if the input voltage is 0 the output current will be 0. Figure E.2 shows how the qualitative models of resistors and switches described above can be organized so as to reproduce this behavior.

Figure E.2: TTL Inverter as Modeled with TINT



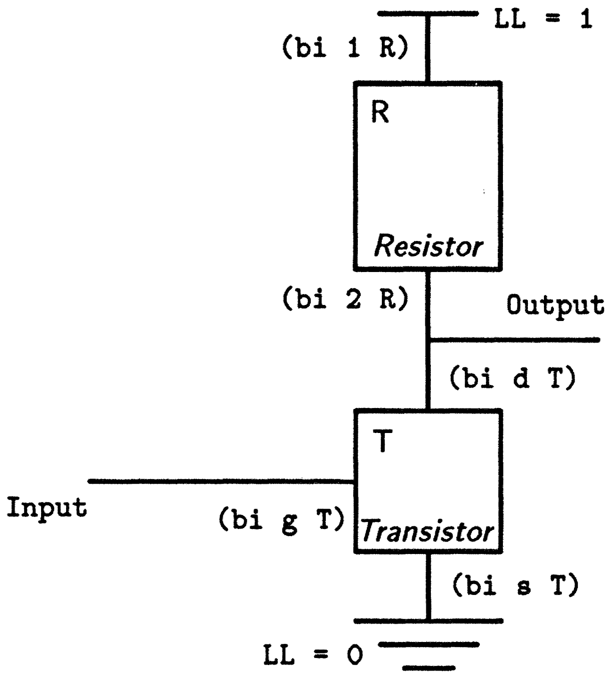
- If (ll Input) is 0, then the difference between (ll (bi 1 R)) and (ll (bi 2 R)) is 1, so (qci 2 R) is -. Hence, current is flowing out of the resistor and back towards the gate driving this one. The switch is shut and so (ll (bi 1 T)) has the same value as (ll (bi 2 T)), hence (ll Output) is 0.

- If (qci (bi 2 R)) is 0, then (ll (bi 2 R)) must be pulled up to 1, the same as (ll (bi 1 R)). This makes the switch open, so (qci (bi 2 T)) is 0, and the gate being driven will make (ll Output) be 1.

Similarly, Figure E.3 shows the model of an nMOS inverter in TINT. In nMOS, the current normally flowing into the device from the input is 0 and so likewise for the current into the output.

- If (ll Input) is 1 then the switch is shut, so (ll (bi 2 T)) has the same value as (ll (bi 1 T)), hence (ll Output) is 0.

Figure E.3: nMOS Inverter as Modeled with TINT



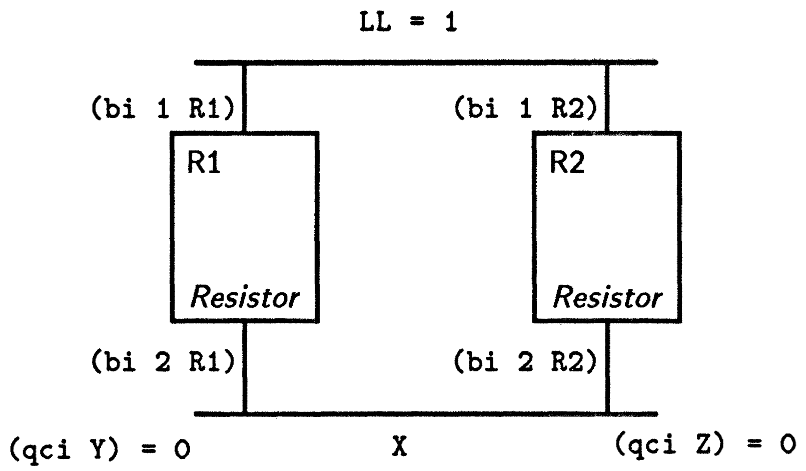
- If `(ll Input)` is `0` then the switch is **open**, so `(qci (bi 2 T))` is `0`. Hence `(qci (bi 2 R))` is `0` and hence `(ll (bi 2 R))` must be `1`.

Similar models apply to NAND and NOR gates in both technologies. A tristate driver in either technology can be described as a nMOS inverter with a transistor interposed between the pullup resistor and the output node.

The disadvantage of this level of detail is that while the the digital model allows the behavior of a given group of boolean gates to be easily predicted using straightforward local propagation, this cannot be done in general in the switch model. At every signal fanout in TTL or wired-OR in nMOS, local propagation stalls, and the solutions to that problem all have unfortunate side effects. This is a standard problem with local propagation schemes; what is different about this case is that it is guaranteed to be ubiquitous at the switch level of detail.

For example, the rules shown so far cannot deduce that the node X in Figure E.4 must have logic-level 1, nor that the currents into the resistors must be 0. This is because either one of those facts must be known before the other can be deduced; this is termed an impasse.

Figure E.4: Impasse Example



One solution is to enumerate the possible values of logic-level at X (there are only two) in hopes that all but one can be ruled out. In this case, 0 is inconsistent because it would require the sum of currents into X to be positive. Thus, the logic-level must be 1. This is a terrible solution in general, because it can lead to combinational explosion among choices made for different quantities over different time intervals. TINT does not use this solution.

A second solution is to recognize that R1 and R2 are in parallel, and since their resistances are positive then the resistance between the high voltage and X is positive too. In effect, there is just a single resistor between the two nodes — a *slice* [Sussman77] [Sussman80]. In BASIL terminology, there is a functional component including R1, R2, and etch X, and its behavior rule recognizes the above situation and just assigns the logic-level 1 to X. TINT uses this solution in the Console Controller Board examples where there happen to be two or more resistor components pulling up a single circuit node.

A third solution is to rely on the intended direction of signal flow between the components and assume that no fault will cause that to be violated. This is a way of using the switch model for just those components that really need it (resistors and switches) while retaining the simpler unidirectional digital model for everything else. It is the solution that TINT uses everywhere that the intended signal flow is unidirectional. Shorting faults will be misdiagnosed as multiple faults among the shorted components, since the effect of shorts is to cause current to go places where it was not intended to go. In TTL it is usually the case that each node is driven by only one component, and if that component does not hold the node to logic-level 0, some other component pulls it up to logic-level 1. The component driving the node can simply be modeled as if it pulls the node to 1 itself. Thus the signal flow appears to be unidirectional. The behavior of TTL components with respect to qualitative currents is thus approximated using the following rules. First, if there is no current into the input of a TTL component then the node is pulled up to 1:

```
  If[1] ?x is a TTL component
  and   [thru ?l1 ?u1 (mode ?x) normal]
  and   [thru ?l2 ?u2 (qci (in ?input ?x)) 0]
  and   ?input is not either PWR or GND
  and   (overlap (?l1 ?u1) (?l2 ?u2))
  Then  [thru (max ?l1 ?l2) (min ?u1 ?u2)
              (ll (in ?input ?x)) 1]
```

Second, it has been assumed that if a component is not pulling its output node down, then it will be pulled up to 1; hence if there is no current flowing through a pin intended to be a TTL output, then the logic-level at the node is 1:

```
  If   ?x is a TTL component
  and  [conn (pin ?i ?c) (hole ?m ?e) (out ?o ?x)]
  and  [thru ?l ?u (qci (hole ?m ?e)) 0]
  Then [thru ?l ?u (ll (hole ?m ?e)) 1]
```

[1]This trigger pattern is implemented using a separate rule for each type of TTL component.

These latter two rules are used for all the TTL components in the implemented model of the Console Controller Board; the board has a two CMOS chips and for the time being they are modeled as if they were TTL as well. Where resistors appear in wired-or structures and as pullups for buttons and switches, the digital current model is used, and since it results in deductions being made about logic-levels it meshes smoothly with the standard digital model.

Bibliography

[Abelson85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[AbuHanna88] A. Abu-Hanna and Y. Gold. An Integrated, Deep-Shallow Expert System for Multi-Level Diagnosis of Dynamic Systems. Technical Report 504, Technion – Israel Institute of Technology, Haifa 32000, Israel, March 1988.

[Allen83] J. Allen. Maintaining Knowledge about Temporal Intervals. *Comm. of the ACM*, 26(11):832–843, 1983.

[Allen84] J. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23(2):123–154, July 1984.

[Batali81] J. Batali. An Introduction to DPL. Memo 598, MIT Artificial Intelligence Lab, 1981.

[Bobrow85] D. Bobrow, editor. *Qualitative Reasoning about Physical Systems.* MIT Press, Cambridge, MA, 1985.

[Brown76] A. Brown. Qualitative Knowledge, Causal Reasoning, and the Localization of Failures. Technical Report 362, MIT Artificial Intelligence Lab, 1976.

[Brown82] J. S. Brown, R. Burton, and J. de Kleer. Pedagogical, Natural Language, and Knowledge Engineering Issues in SOPHIE I, II, and III. In D. Sleeman and J. S. Brown, editors, *Intelligent Tutoring Systems*, pages 227–282. Academic Press, New York, 1982.

[Cantone83] R. Cantone, F. Pipitone, W. Lander, and M. Marrone. Model-based Probabilistic Reasoning for Electronics Troubleshooting. In *Proc. 8th Int. Joint Conf. on Artificial Intelligence*, pages 207–211, Karlsruhe, West Germany, August 1983.

[Dague87] P. Dague, O. Raiman, and P. Deves. Troubleshooting: When Modeling is the Difficulty. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 600–605, Seattle, WA, August 1987.

[Davis83] R. Davis and H. Shrobe. Representing the Structure and Behavior of Digital Hardware. *IEEE Computer*, pages 75–82, October 1983.

[Davis84] R. Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24(1):347–410, 1984. Also in *Qualitative Reasoning about Physical Systems*, Bobrow (ed.), MIT Press, Cambridge, MA 1985.

[deKleer76] J. de Kleer. Local Methods for Localizing Faults in Electronic Circuits. Memo 394, MIT Artificial Intelligence Lab, 1976. Out of print.

[deKleer78] J. de Kleer. Causal and Teleological Reasoning in Circuit Recognition. Technical Report 529, MIT Artificial Intelligence Lab, September 1979.

[deKleer84] J. de Kleer and J. S. Brown. A Qualitative Physics Based on Confluences. *Artificial Intelligence*, 24(1):7–84, 1984. Also in *Qualitative Reasoning about Physical Systems*, Bobrow (ed.), MIT Press, Cambridge MA 1985.

[deKleer86a] J. de Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[deKleer86b] J. de Kleer and B. Williams. Back to Backtracking: Controlling the ATMS. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 910–917, Philadelphia, PA, August 1986.

[deKleer87] J. de Kleer and B. C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97–130, April 1987.

[Dean87] T. Dean and D. McDermott. Temporal Data Base Management. *Artificial Intelligence*, 32(1):1–56, April 1987.

[Feldman88] Y. A. Feldman and C. Rich. Pattern-Directed Invocation with Changing Equalities. Memo 1017, MIT Artificial Intelligence Lab, May 1988.

[First82] M. B. First, B. J. Weimer, S. McLinden, and R. A. Miller. LOCALIZE: Computer-Assisted Localization of Peripheral Nervous System Lesions. *Computers and Biomedical Research*, 15(6):525–543, December 1982.

[Friedman83] L. Friedman. Diagnosis Combining Empirical and Design Knowledge. Technical Report JPL D-1328, Jet Propulsion Laboratory, California Institute of Technology, December 1983.

[Geffner86] H. Geffner and J. Pearl. Distributed Diagnosis of Systems with Multiple Faults. Technical Report CSD-860023, Cognitive Systems Laboratory, UCLA Computer Science Department, Los Angeles, CA 90024, December 1986.

[Genesereth84] M. Genesereth. The Use of Design Descriptions in Automated Diagnosis. *Artificial Intelligence*, 24(1):411–436, 1984. Also in *Qualitative Reasoning about Physical Systems*, Bobrow (ed.), MIT Press, Cambridge MA 1985.

[Ginsberg86] M. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35–80, December 1986.

[Gorry73] G. A. Gorry, J. P. Kassirer, A. Essig, and W. B. Schwartz. Decision Analysis as the Basis for Computer-Aided Management of Acute Renal Failure. *American Journal of Medicine*, 55:473–484, October 1973.

[Hall87] R. Hall, R. Lathrop, and R. Kirk. A Multiple Representation Approach to Understanding the Time Behavior of Digital Circuits. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 799–803, Seattle, WA, August 1987.

[Hamscher84] W. C. Hamscher and R. Davis. Diagnosing Circuits with State: An Inherently Underconstrained Problem. In *Proc. 4th National Conf. on Artificial Intelligence*, pages 142–147, Austin, TX, August 1984.

[Hamscher87] W. C. Hamscher and R. Davis. Issues in Model-Based Troubleshooting. Memo 893, MIT Artificial Intelligence Lab, March 1987.

[Hanks86] S. Hanks and D. V. McDermott. Default Reasoning, Nonmonotonic Logics, and the Frame Problem. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 328–333, Philadelphia, PA, August 1986.

[Intel86] Intel. *Intel Microcontroller Handbook.* Intel Corporation, Santa Clara, CA, 1986.

[Kahn77] K. Kahn and G. A. Gorry. Mechanizing Temporal Knowledge. *Artificial Intelligence*, 9(1):87–108, August 1977.

[Kohane87] I. S. Kohane. Temporal Reasoning in Medical Expert Systems. Technical Report 389, MIT Lab. for Computer Science, May 1987.

[Kramer87] G. A. Kramer. Incorporating Mathematical Knowledge into Design Models. In J. S. Gero, editor, *Expert Systems in Computer-Aided Design*, pages 229–265. Elsevier Science Publishers B. V., Amsterdam, 1987.

[Kuipers84] B. J. Kuipers and J. P. Kassirer. Causal Reasoning in Medicine: Analysis of Protocol. *Cognitive Science*, 8:363–385, 1984.

[Kulikowski82] C. A. Kulikowski and S. M. Weiss. Representation of Expert Knowledge for Consultation: The CASNET and EXPERT Projects. In P. Szolovits, editor, *Artificial Intelligence in Medicine*, pages 21–56. Westview Press, Boulder, CO, 1982.

[Ladkin87] P. Ladkin. The Completeness of a Natural System for Reasoning with Time Intervals. In *Proc. 10th Int. Joint Conf. on Artificial Intelligence*, pages 462–467, Milan, Italy, 1987.

[Lifschitz87] V. Lifschitz. Formal Theories of Action (Preliminary Report). In *Proc. 10th Int. Joint Conf. on Artificial Intelligence*, pages 966–972, Milan, Italy, August 1987.

[Long86] W. J. Long, S. Naimi, M. G. Criscitiello, and R. Jayes. Using a Physiological Model for Prediction of Therapy Effects in Heart Disease. In *Computers in Cardiology*, Cambridge, MA, 1986.

[McAllester80a] D. A. McAllester. The Use of Equality in Deduction and Knowledge Representation. Technical Report 550, MIT Artificial Intelligence Lab, January 1980.

[McAllester80b] D. A. McAllester. An Outlook on Truth Maintenance. Memo 551, MIT Artificial Intelligence Lab, August 1980.

[McCarthy69] J. M. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In D. Michie and B. Meltzer, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Scotland, 1969. Also in *Readings in Artificial Intelligence*, B. L. Webber and N. J. Nilsson (eds.), Tioga Press, 1981.

[McDermott82] D. V. McDermott. A Temporal Logic for Reasoning about Processes and Plans. *Cognitive Science*, 6(2):101–155, April 1982.

[Milne85] R. Milne. Fault Diagnosis through Responsibility. In *Proc. 9th Int. Joint Conf. on Artificial Intelligence*, pages 423–425, Los Angeles, CA, August 1985.

[Minsky75] M. Minsky. A Framework for Representing Knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, New York, NY, 1975.

[Moszkowski82] B. Moszkowski. A Temporal Logic for Multi-Level Reasoning about Hardware. Technical Report STAN-CS-82-952, Stanford University Artificial Intelligence Lab., 1982.

[Pan84] J. Pan. Qualitative reasoning with Deep-level Mechanism Models for Diagnoses of Mechanism Failures. In *Proc. 1st Conf. on A.I. Applications*, pages 295–301, Denver, CO, 1984.

[Patil81] R. S. Patil. Causal Representation of Patient Illness for Electrolyte and Acid-Base Diagnosis. Technical Report 267, MIT Lab. for Computer Science, October 1981.

[Pauker76] S. G. Pauker, G. A. Gorry, J. P. Kassirer, and W. B. Schwartz. Towards the Simulation of Clinical Cognition: Taking a Present Illness by Computer. *American Journal of Medicine*, 60:981–996, June 1976.

[Pople82] H. E. Pople. Heuristic Methods for Imposing Structure on Ill-structured Problems: The Structuring of Medical Diagnostics. In P. Szolovits, editor, *Artificial Intelligence in Medicine*, pages 119–190. Westview Press, Boulder, CO, 1982.

[Reggia83] J. A. Reggia, D. S. Nau, and P. Wang. Diagnostic Expert Systems Based on a Set Covering Model. *Int. Journal of Man-Machine Studies*, 19(5):437–460, November 1983.

[Reiter87] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–96, April 1987.

[Roth67] J. P. Roth, W. G. Bouricius, and P. R. Schneider. Programmed Algorithms to Compute Tests to Detect and Distinguish between Failures in Logic Circuits. *IEEE Transactions on Electronic Computers*, EC-16(1):567–580, 1967.

[Rowley87] S. Rowley, H. Shrobe, R. Cassels, and W. C. Hamscher. Joshua: Uniform Access to Heterogeneous Knowledge Structures, or, Why Joshing is Better than Conniving or Planning. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 45–52, Seattle, WA, 1987.

[Russ86] T. A. Russ. A System for Using Time Dependent Data in Patient Management. In *MEDINFO 86: Proceedings of the 5th Conference on Medical Informatics*, pages 165–169, Washington, DC, October 1986.

[Scarl85] E. Scarl, J. R. Jamieson, and C. I. Delaune. A Fault Detection and Isolation Method Applied to Liquid Oxygen Loading for the Space Shuttle. In *Proc. 9th Int. Joint Conf. on Artificial Intelligence*, pages 414–416, Los Angeles, CA, 1985.

[Shirley83] M. H. Shirley and R. Davis. Generating Distinguishing Tests based on Hierarchical Models and Symptom Information. In *Proc. Int'l Conference on Computer Design*, 1983.

[Shoham86] Y. Shoham. Chronological Ignorance: Time, Nonmonotonicity, Necessity, and Causal Theories. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 389–393, Philadelphia, PA, August 1986.

[Shoham87] Y. Shoham. Temporal Logics in AI: Semantical and Ontological Considerations. *Artificial Intelligence*, 33(1):89–104, September 1987.

[Shortliffe76] E. H. Shortliffe. *MYCIN: Computer-Based Consultations in Medical Therapeutics.* American Elsevier, New York, 1976.

[Simmons83] R. G. Simmons. Representing and Reasoning about Change in Geologic Interpretation. Technical Report 749, MIT Artificial Intelligence Lab, December 1983.

[Steele84] G. L. Steele. *Common LISP: The Language.* Digital Equipment Corporation, 1984.

[Sussman77] G. J. Sussman. SLICES: At the Boundary between Analysis and Synthesis. Memo 433, MIT Artificial Intelligence Lab, 1977. This memo is out of print.

[Sussman80] G. J. Sussman and G. L. Steele. Constraints: A Language for Expressing Almost-hierarchical Descriptions. *Artificial Intelligence*, 14(1):1–40, January 1980.

[Szolovits78] P. Szolovits and S. G. Pauker. Categorical and Probabilistic Reasoning in Medical Diagnosis. *Artificial Intelligence*, 11:115–144, 1978.

[Valdes86] R. Valdes-Perez. Spatio-Temporal Reasoning and Linear Inequalities. Memo 875, MIT Artificial Intelligence Lab, May 1986.

[Valdes87] R. Valdes-Perez. The Satisfiability of Temporal Constraint Networks. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 256–260, Seattle, WA, August 1987.

[VanBaalen88] J. Van Baalen and R. Davis. Overview of an Approach to Representation Design. In *Proc. 7th National Conf. on Artificial Intelligence*, pages 392–397, Minneapolis, MN, August 1988.

[Vilain86] M. Vilain and H. Kautz. Constraint Propagation Algorithms for Temporal Reasoning. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 377–382, Philadelphia, PA, August 1986.

[Weise86] D. Weise. Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits. Technical Report 978, MIT Artificial Intelligence Lab, August 1986.

[Weld86] D. S. Weld. The Use of Aggregation in Qualitative Simulation. *Artificial Intelligence*, 30(1):1–34, October 1986.

[Williams86] B. C. Williams. Doing Time: Putting Qualitative Reasoning on Firmer Ground. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 105–112, Philadelphia, PA, August 1986.