Technical Report 707

Using Structural and Functional Information in Diagnostic Design

Walter Hamscher

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

USING STRUCTURAL AND FUNCTIONAL INFORMATION IN DIAGNOSTIC DESIGN

by

Walter Hamscher

(c) Massachusetts Institute of Technology
June 1983

Revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on 13 May 1983 in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

We wish to design a diagnostic for a device from knowledge of its structure and function. The diagnostic should achieve both *coverage* of the faults that can occur in the device, and should strive to achieve *specificity* in its diagnosis when it detects a fault.

A system is described that uses a simple model of hardware structure and function, representing the device in terms of its internal primitive functions and connections. The system designs a diagnostic in three steps. First, an extension of path sensitization is used to design a test for each of the connections in the device. Next, the resulting tests are improved by increasing their specificity. Finally the tests are ordered so that each relies on the fewest possible connections.

We describe an implementation of this system and show examples of the results for some simple devices.

Thesis supervisor: Randall Davis

Title: Assistant Professsor of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

I would like to express my thanks to a few of the people who encouraged and aided me in this work:

Randy Davis, my thesis supervisor, for 2.3 person-years of superhuman patience, wisdom, insight, and encouragement.

Everyone associated with the Hardware Troubleshooting Group, particularly Howard Shrobe, Steve Polit, Karen Wieckert, Mark Shirley, Harold Haig, Ramesh Patil, and Dan Carnese, for 9.2 person-years of questions, ideas, and criticism.

Jon Sieber, Chris Terman, Rich Zippel, Oded Feingold, Jeff Arnold, Geof Cooper, Paul Rando, Jim Kulp, and Mike Konopik, who contributed in various ways to keeping our group's machine running and our software current, for a jillion person-hours of help.

My housemates Debbie, Kate, Elissa, Colin, and Danny, for 10 person-years of comradeship.

Mom and Dad, for 49 person-years of love.

CONTENTS

1. Introduction 1

2. Previous Work in Test Generation 4

    2.1 Path Sensitization and the D-Algorithm 4
    2.2 Functional Testing 6

3. Models 8

    3.1 Multiplexers 8
    3.2 Devices 8
    3.3 Paths 13
    3.4 The Fault Model 14
    3.5 Inquiries 15
    3.6 Conditions in the Information Path Model 16

4. Overview of the Diagnostic Generation Procedure 18

    4.1 The Inquiry Design Phase 18
    4.2 The Inquiry Improvement Phase 21
    4.3 Test Ordering 22
    4.4 Code Generation 22

5. Inquiry Design Rules and Their Use 23

    5.1 Behavior Rules 23
    5.2 Sensitization Rules 24
    5.3 Goal Rules 26
    5.4 Condition Rules 27
    5.5 An Example Of An Inquiry Design 29
    5.6 Propagation Through Time 32
    5.7 Summary 34

6. The Inquiry Improvement Phase 35

    6.1 Elimination of "OR" Conditions Using the SPFA 35
    6.2 Collaboration 36
    6.3 Transforming Conditions Using Implementation Information 37
    6.4 Improving Inquiry Efficiency 38

7. The Ordering Phase 40

7.1 Test Aggregation 40
7.2 Rules for Pairwise Ordering of Tests 41
7.3 Ordering the Multiplexer Tests 44
7.4 Local Preordering 45

8. Implementation 47

8.1 Rules 47
8.2 Avoiding Or-Conditions 48
8.3 Avoiding Inquiry Designs 49
8.4 Inquiry Design as a Search Problem 49

9. Future Directions 51

9.1 Limitations of the Inquiry Design Methodology 51
9.2 Inquiry Design With State Devices 55
9.3 Diagnostic Design as a Problem 55

10. Conclusion 57

Appendix I. Notation 59

FIGURES

Fig. 1. A 4x4 Memory 1
Fig. 2. An IO Bus 2
Fig. 3. Path Sensitization Example 4
Fig. 4. Examples of D-cubes for a 2-input NAND 5
Fig. 5. Functional Test Example 6
Fig. 6. Boolean Multiplexer 9
Fig. 7. Tristate Multiplexer 9
Fig. 8. Gate 10
Fig. 9. Junction 10
Fig. 10. Selector 10
Fig. 11. Information Path Model of a 3-way Multiplexer 11
Fig. 12. Fanout 11
Fig. 13. Memory 11
Fig. 14. Two Implementations of the Junction Primitive 12
Fig. 15. Two Implementations of the Gate Primitive 13
Fig. 16. Design Information in the Multiplexer 13
Fig. 17. An Inquiry 15
Fig. 18. Three Tristates implementing a Gate primitive 17
Fig. 19. A Behavior Rule 18
Fig. 20. A Sensitization Rule 19
Fig. 21. A Goal Rule 19
Fig. 22. A Condition Rule 19
Fig. 23. Focus Assignments for the Inquiry [DO1 ? d2] 20
Fig. 24. Multiplexer Assignments for the Inquiry [DO1 ? d2] 21
Fig. 25. Behavior Rule GB-1 23
Fig. 26. Behavior Rule GB-2 24
Fig. 27. Behavior Rule SB-1 24
Fig. 28. Behavior Rule JB-1 24
Fig. 29. Sensitization Rule GS-1 25
Fig. 30. Sensitization Rule GS-2 25
Fig. 31. Sensitization Rule JS-1 26
Fig. 32. Sensitization Rule GS-3 26
Fig. 33. Prototype Goal Rule for Backward Propagation 26
Fig. 34. Prototype Goal Rule for Forward Propagation 27
Fig. 35. Goal Rule for Forward Propagation with Multiple Outputs 27
Fig. 36. Condition Rule GC-1 28
Fig. 37. Condition Rule GC-2 28
Fig. 38. Condition Rule GC-3 29
Fig. 39. Condition Rule GC-4 30
Fig. 40. Goal Assignments in the Multiplexer 30
Fig. 41. Value Assignments in the Multiplexer 31

Fig. 42. Condition Assignments in the Multiplexer 31
Fig. 43. Propagations at Time t 33
Fig. 44. Propagations at Time t-1 34
Fig. 45. Condition Resulting from Or-Conditions on Reconvergent Paths 36
Fig. 46. Two Inquiries for a Simple Device 38
Fig. 47. Multiplexer Tests with XI-conditions Removed 44
Fig. 48. Ordering of Multiplexer Tests after Rules 1, 2 and 3 44
Fig. 49. Local Application Of Ordering Rules to a Gate 46
Fig. 50. Conditions Resulting from Reconvergent Paths 48
Fig. 51. One of Three Inquiries for [DO ? di] 51
Fig. 52. Sensitization Rule GS-5 52
Fig. 53. Result of Applying GS-5 54
Fig. 54. Assignment of Conditions After Choosing (A = d1) 54
Fig. 55. Assignments in an Addressable Memory at Time t 56

1. Introduction

Diagnostics are programs that are run to test a device when it comes off the assembly line and when there is evidence that the device has failed in the field. Their purpose is to check for the presence of physical faults by exercising all the functions of the component and looking for evidence of misbehavior. Diagnostics should achieve both *coverage* and *resolution*. A diagnostic has *coverage* if it detects all the faults that could occur in the device under a particular fault model. *Resolution* is the specificity with which the diagnostic can identify the parts that could be responsible when a fault is detected.

We wish to design a diagnostic for a hardware component, given a structural and functional description of it. Designing diagnostics from such descriptions is an important task, and yet is still done largely by hand. Our goal is to automate this process with a program that has an understanding of structure and function comparable to that of human designers.

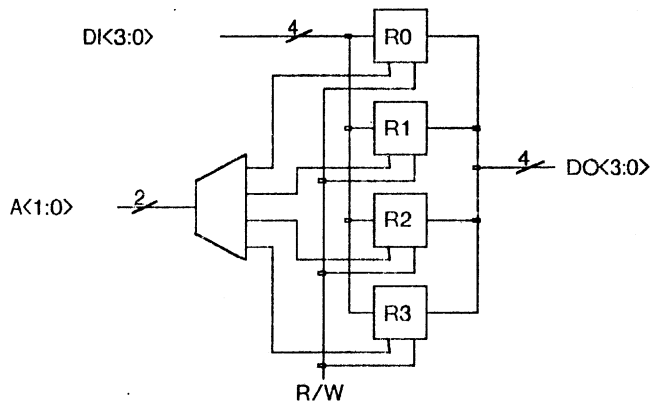
Some parts of this problem are well understood. Languages for describing hardware structure and function have existed for some time. Test generation systems have already been designed for parts of computers, principally by making use of well-known algorithms such as the D-algorithm. Unfortunately, the D-algorithm (like others) adheres rigidly to the stuck-at fault model, uses the "single-fault" assumption, and rarely uses any description level beyond that of single gates. We have yet to see a system that accepts a structural and functional specification of a system and by reasoning solely from this description produces a complete and effective set of diagnostics.

In exploring this problem we have considered the performance of human experts at this task and have been impressed by how much of the design is done knowing relatively little about the device in question. Consider for example how a diagnostic writer would design a diagnostic for the 4x4 memory shown in Figure 1.

She assumes initially that the fault will be one of the wires or cells stuck at 1 or 0, and assumes that there is only a single point of failure (knowing that if necessary she can modify her diagnostic later to cover faults that violate those assumptions). She can now plan a diagnostic for this device by knowing only that the address selects one of the data inputs and routes its data to or from a single register. That much knowledge tells her that she should

---

Fig. 1. A 4x4 Memory



verify the addressing lines before testing the data inputs, and that she can test the data lines independently of any single register by iterating over several values of the address. She would organize the diagnostic into phases:

(1) Test whether the data input and output of the memory transmit data correctly.

(2) Test each whether each register can be addressed.

(3) Test whether data can be correctly stored at each register.

The plan shows attention to both coverage and resolution. It achieves coverage of faults by testing the address, data input and output, and the individual registers. It achieves resolution by testing one function at a time and relying only on functions that have been previously tested.

A similar diagnostic could be written for a 4x4 multiplexer: test the output, then the address, then the individual inputs. The similarity arises from the similarity of function between the memory and the multiplexer: each uses an address to route data internally.

Consider also designing a diagnostic for a processor's IO bus to which are attached controllers for terminals, a lineprinter, and a local network (Figure 2).

Again there are similarities of strategy, because the *information transmission behavior* of the IO bus is similar to that of a memory in spite of the difference in scale. The bus address routes information from the processor to the selected controller and back. This diagnostic falls into phases similar to that for the memory:

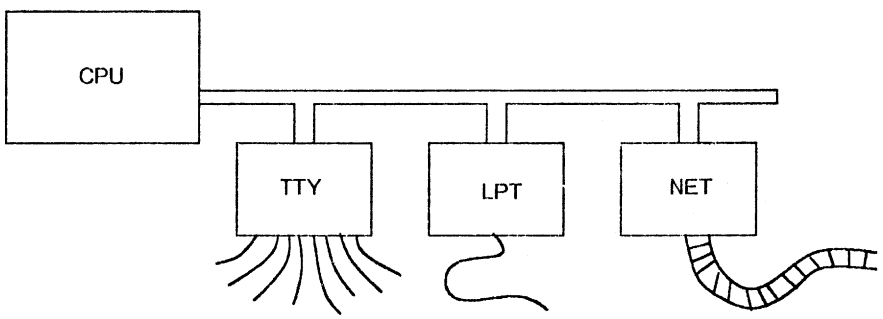
(1) Test whether data is being transmitted correctly by the bus.

(2) Test each controller to see whether it responds to its address.

(3) Test whether data can be correctly transmitted to and from each controller.

There is a common theme to these examples: we were able to plan the overall structure of the diagnostics in spite of very limited information. This is the competence that our system tries to capture.

Fig. 2. An IO Bus



To accomplish this it relies on a simple model of hardware structure and behavior and uses some general assumptions and principles of diagnostic design. With this foundation it is able to design a series of tests that provide coverage and resolution. The simple model can then be augmented with specific knowledge about the device, and the resulting tests can be refined using this knowledge.

This simple model of hardware, which we call the *information path* model, is intended to capture the ability of humans to plan diagnostics without knowing very much about the hardware implementation. The representation is used to determine what tests need to be run and what dependencies exist among those tests.

Our system uses these principles to design a diagnostic in three steps. First, an extension of path sensitization is used to design a test for each of the connections in the device. Next, the resulting tests are improved by increasing their specificity. Finally, the tests are ordered so that each relies on the fewest possible untested connections. We describe a program which implements this system. The program treats the first phase as a search problem in the space of possible tests for a given device, and treats the second and third steps as a series of straightforward reorganization operations on sets.

2. Previous Work in Test Generation

Until recently, most efforts in automated test generation have focused on gate-level representations of combinatorial circuits and have concentrated on achieving coverage rather than resolution. The methodology typically employed is *path sensitization*. Experience with path sensitization, as described below, seems to demonstrate that (1) it is most successful when gate-level descriptions are used, but this is computationally expensive; (2) if more abstract functional descriptions are used, a lack of correspondence between those functional descriptions and their hardware implementations reduces both the coverage and resolution of the resulting test sequences.

2.1 Path Sensitization and the D-Algorithm

Path sensitization relies on two basic concepts: a fault must be *sensitized* and the result *propagated* to an output. A fault is sensitized when the presence of the fault will make a difference in the behavior of the device. For example, if a wire is stuck at zero (sa-0) and we try to force it to 1, then the fault is sensitized. A result is *propagated* to an output so that the effect of the fault will be visible to an observer. This is done by choosing the remaining inputs of the device so that the output of the faulty part will be propagated to an observable output.

Consider the simple circuit in Figure 3. To sensitize wire Y to the presence of a sa-1 fault, choose inputs W and X to be 1. To propagate the sa-1, we choose input V to be 1. In this way, the observable output Z will be 0 if the fault is present, and 1 otherwise. Path sensitization takes its name from the fact that a path of sensitized wires results from sensitizing a single fault.

Note that a sa-0 fault on the wires W, X, or Z would cause the circuit to appear faulty. Thus W, X and Z are also sensitized to sa-0 faults.

The best-known algorithm for path sensitization is the *D-algorithm* [Roth 67], which uses exhaustive search to generate tests for sa-0 and sa-1 faults in combinatorial circuits. The algorithm takes its name from its use of *D*, meaning *discrepancy*, as an abstraction of stuck-at-zero, and likewise *~D* as an abstraction of stuck-at-one. Each primitive boolean gate has a *D-cube* that defines the propagation behavior of the gate. These cubes are truth tables extended to include the symbols D and ~D. Two examples of D-cubes are shown in Figure 4.

Fig. 3. Path Sensitization Example

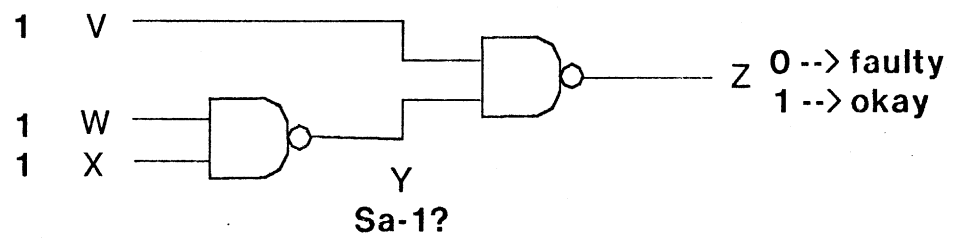


Fig. 4. Examples of D-cubes for a 2-input NAND

```
------+----          ------+---
 1  1 | ~D           1 ~D | D
------+----          ------+---
```

The first cube says that to sensitize ~D on the output of a 2-input NAND, make both inputs 1. The second cube says that to propagate ~D from an input, make the gate's other input 1 and test the output for sa-0. In the previous example, Y would be sensitized to ~D by using the cube on the left, and ~D would be propagated using the cube on the right.

A propagation D-cube typically requires choices of input values for the test. Because inconsistencies occur and backtracking must be done, the algorithm is essentially an exhaustive search through alternative inputs. Detailed descriptions of the D-algorithm can be found in [Breuer 76] and [Roth 80].

Extensive effort has been devoted to showing ways that path sensitization algorithms can been extended to handle special classes of circuits efficiently and deal with multiple faults. The principal result of efforts to handle multiple faults has been the observation that any test that detects a single stuck-at fault will also detect some (but not all) multiple stuck-at faults.

Efforts have also been made to extend the D-algorithm to distinguish between faults, i.e. to achieve resolution. But achieving resolution in a k-input combinatorial circuit, even using an extension of the D-algorithm, involves trying all $2\uparrow k$ combinations of inputs, which is exhaustive testing.

The D-algorithm has the virtue that it is guaranteed to generate a complete set of tests for any combinatorial circuit, where "complete" is taken to mean that the tests can find all the faults in a particular class, under a strong set of assumptions. Because sensitizing a fault in a combinatorial circuit is NP-complete,[1] generating the tests is computationally expensive. As circuits have become larger, the efficiency of the generation technique has become a concern.

Heuristic methods may be used to improve the efficiency of test generation. [Rutman 72] describes a system in which achieving a 1 or 0 on a particular wire is regarded as a subproblem of a sensitization problem. In the example above, achieving a 1 on wire Y is a problem that spawns the subproblems of achieving 1's on W and X and a 1 on V. The system estimates the difficulty of all such subproblems once for the entire circuit, and performs best-first search to do each path sensitization.

Using higher-level models of components is also expected to improve the efficiency of test generation algorithms. [Breuer 79] demonstrates that sensitization and propagation can be done for devices such as adders and shifters. Currently the complexity of generating methods for such devices appears to be a principal limitation. The simplicity with which propagation D-cubes can be created for boolean devices partially accounts for the success of

1. Given a k-input conjunctive normal form expression, build a k-input, 1-output circuit from AND and OR gates that implements it. Sensitize the output to a sa-0 fault. If that can be done, the expression is satisfiable.

the D-algorithm, which is thereby closely tied to the "bits and gates" level of hardware description.

2.2 Functional Testing

Some recent work on testing [Lai 81] represents a departure from traditional path sensitization. Lai applied path sensitization to a functional description of a PDP-8 CPU. The program produces *functional tests*, where functional testing is claimed to subsume design validation, acceptance testing, and field diagnosis. The fault model used to generate the tests was a *path fault*, corresponding to single-bit stuck-ats on the connections between functional primitives.

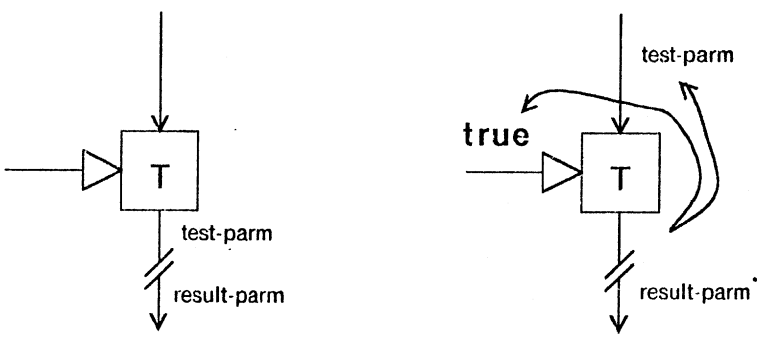
The example in Figure 5 shows how a test is generated for the output of a primitive element of the functional description. The example shows a *true gate*: when the control input at the left is **true**, the input at the top passes through to the output.

The program inserts a functional fault, in this case equivalent to a single bit stuck-at 0, on the output of the gate. It sensitizes the output path by placing **true** on the control, and a test parameter representing "all 1's" on the top input. The output appearing from the gate is a parameter representing the result of the test.

Propagations occur across primitives representing functional abstractions such as decoders and adders. The program generates descriptions of test cases, and test programs are coded by hand from the automatically generated descriptions.

Note that because the function paths do not necessarily correspond to physical implementations in the processor, the path fault model mixes the notion of physical faults with functional faults. The performance of the program was evaluated with respect to finding these functional faults and was compared to the diagnostics provided by the manufacturer. The automatically generated tests achieved 96% functional fault coverage, while the manufacturer's diagnostics provided only 92% coverage.

Fig. 5. Functional Test Example



No effort for resolution was made and no claims were made about the specificity of diagnosis achieved by the tests. A single physical fault could cause many tests to fail and a single test failure might have been caused by many physical faults. As a result, the tests are useful for design validation and acceptance testing, but are less useful from a repair perspective.

3. Models

Our approach to achieving coverage and resolution is based on choosing an appropriate level of abstraction and viewing the diagnostic as a collection of primitive tests that can be ordered to increase their resolution. The key point in our selection of a level of abstraction is the notion that devices can be viewed as being composed of primitive functional components connected by *information paths*. We use this model to abstract away from the digital implementation details as much as possible and yet retain the ability to map the designed test sequence back onto the real device when the time comes.

To illustrate these ideas we examine two implementations of a multiplexer. Both implementations can be viewed in terms of common functional primitives, and we describe these primitives. We then explore some of the subtleties of information paths and introduce a simple fault model. Finally we introduce the notion of *inquiries*, which are the fundamental units upon which diagnostics are organized.

3.1 Multiplexers

Consider the 4x2 multiplexers shown in figures 6 and 7. One is built from boolean devices, the other is done with tristates.

In the first multiplexer, the address bits are decoded to select one of four data inputs. The selected input passes through two stages of NAND gates, getting inverted twice. The unselected inputs are blocked by the fact that there is at least one 0 input to each NAND gate; this causes those NAND gates to assert 1. The second stage of NAND gates have all inputs 1 except those selected, so those gates invert the output of the selected gates.

In the tristate multiplexer, the address bits are decoded and select one of the four data inputs. The unselected tristates allow the output to float, while the selected tristate forces the output to be the same as the selected input.

The explanations of the two implementations are similar: the address input selects some data input and that data input goes to the multiplexer output, while the unselected data inputs have no effect on the multiplexer output. This suggests that both multiplexers can be described using the same functional primitives.

3.2 Devices

We use three primitive functional devices to represent the multiplexer: the *Gate*, the *Junction*, and the *Selector*.

The *Gate* has a control input which determines whether its data input is transmitted to its output. If the control input has the value `on`, then the data output will be the same as the data input. If the control input has the value `off`, the data output will be the value *Xg*, which indicates that the output is not sensitive to its data input (we explain the concept of *Xg* further in 3.2.1). The range of values that the control input can have is {`on`,`off`}. The data input can have any value in the set {*d0*,*d1*...*dn-1*}. We will call this set D and note that $n = 2 \uparrow k$, where k

Fig. 6. Boolean Multiplexer

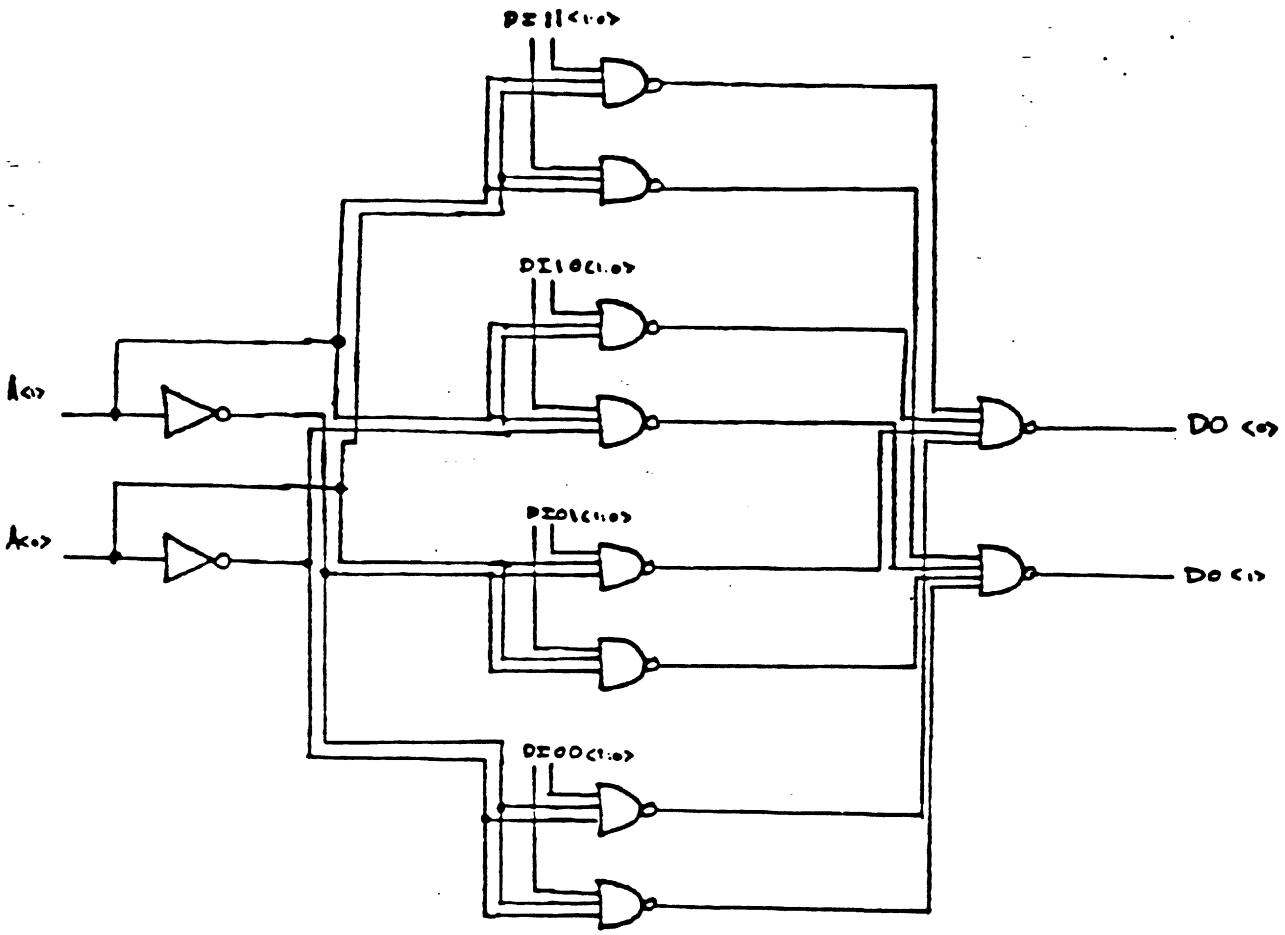
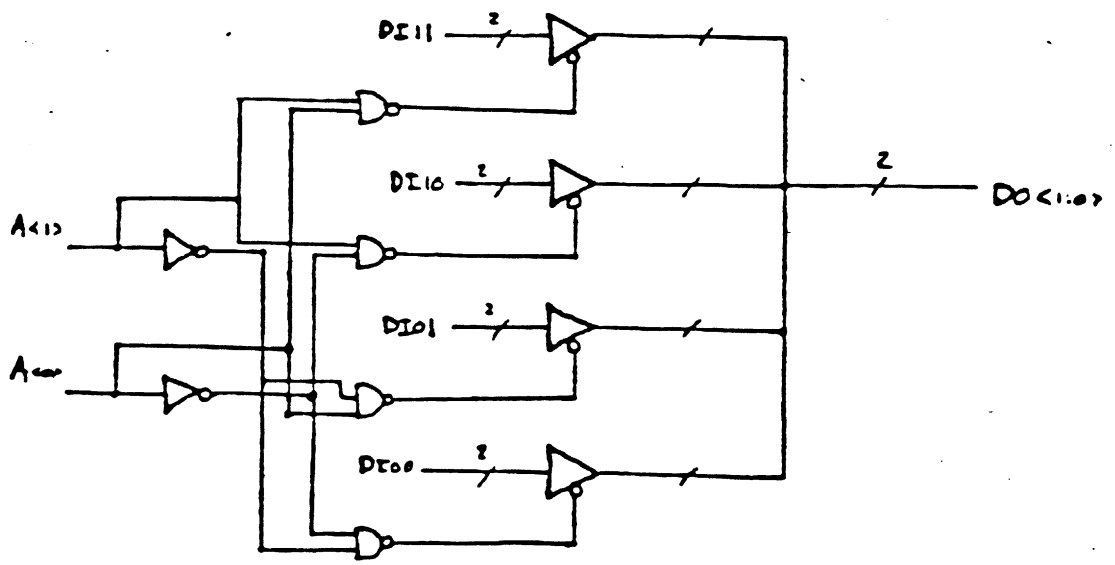


Fig. 7. Tristate Multiplexer



is the width of the path in bits.[2] The output of a gate can have any value in D as well as the value X*g*.

The *Junction* merges several data transmission paths, called fanins. X*j* represents the fanin value that causes the output to be sensitive to other fanins (see 3.2.1). If all but one of the fanins is X*j*, then the output will be the value of that other fanin. For example (Figure 9) if fanin-1 and fanin-2 are X*j* and fanin-3 is d2 then the output is d2. The range of values on the junction's fanins is {X*j*,d*0*,d*1*...d*n-1*}.

The address input of a *Selector* determines which of its outputs will have the value `hi`; the other outputs will have the value `lo`. Thus if the address is d*1* then the 1st output will be `hi` and the other outputs will be `lo`.

The multiplexer that we build from these primitives is shown in Figure 11. For simplicity of presentation, the one we use has only three data paths.

There are currently two other primitives in our language, the *Fanout* and *Memory*.[3] A fanout has a single input that it copies to all its outputs (Figure 12).

Fig. 8. Gate

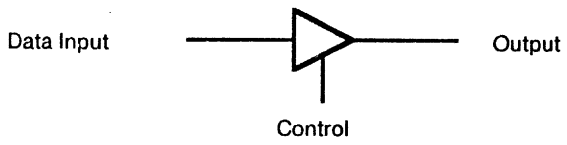


Fig. 9. Junction

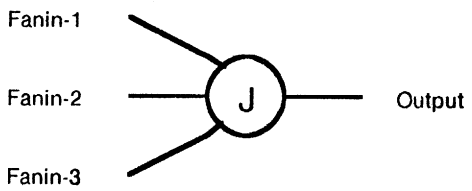
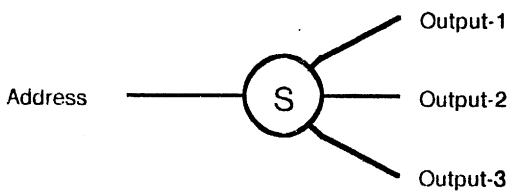


Fig. 10. Selector



2. This is distinct from the D representing a discrepancy in the D-algorithm.
3. These are not used in the multiplexer.

Fig. 11. Information Path Model of a 3-way Multiplexer

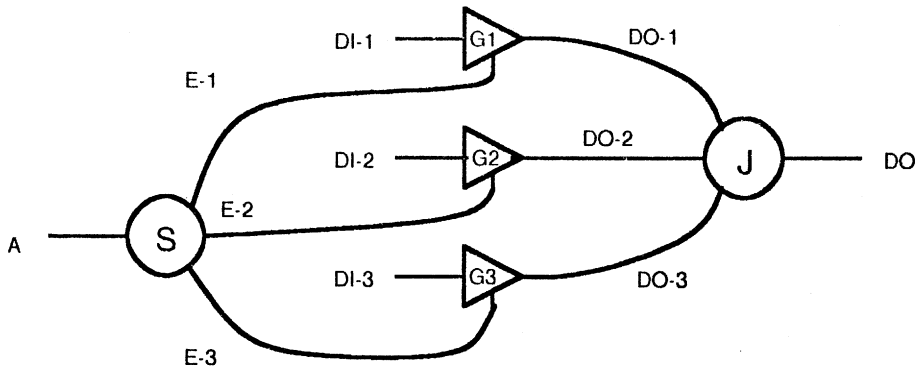
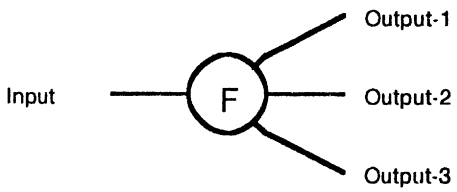


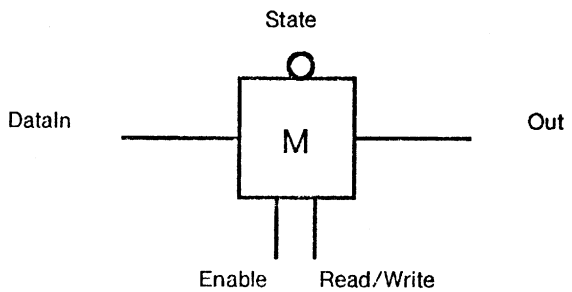
Fig. 12. Fanout



The memory primitive has an enable input, read/write input, data input, a state terminal, and a data output (Figure 13). Data input is transmitted to the state when enable is on and read/write is `write`. The value of the state terminal is preserved over time and is transmitted to the output when enable is on and read/write is `read`. When the enable is `off`, or read/write is `write`, the output is X*m*, indicating that the output is not sensitive to the state.

We use the information-path model with two disclaimers. First, the set of primitives chosen is a preliminary guess; no claim to completeness is intended, and it is expected that the set will in time be expanded. Second, we assume that the information-path model of the device in question is already available. The problem of parsing a hardware structure to extract the information path description is outside the scope of this paper.

Fig. 13. Memory



3.2.1 X Values

Values on paths are an abstraction of digital values on wires. The subscripted X values X*g*, X*j*, and X*m* are also abstractions of digital values, but in addition they capture information about the intended behavior of the path when it is *not* in use: a path transmits an X value when it does not affect, or is unaffected by, values on other paths.

The junction device provides the first example. By definition, the output of a junction is insensitive to a fanin carrying X*j*. The symbol X*j* represents an abstraction of the concept of the null value entering a junction. The easiest way to see this is to consider a 2-way junction implemented as a 2-input OR, and alternatively as a 2-input AND, as shown in Figure 14. In the OR implementation the fanins are A and B; the output is C. In the AND implementation the fanins are F and G; the output is H.

When the junction is implemented as an OR, the null value is 0. This is because when A is 0, C gets the value of B; when B is 0, C gets the value of A. In the AND implementation the null value is 1, because when F is 1, H gets the value of G, and when G is 1, H gets the value of F.

The idea of a null value entering a junction is common to both implementations. We wish to capture this idea without worrying about whether the digital value should be 0 or 1. X*j* represents this abstraction. The example shows that X*j* (a) corresponds to a digital value on a wire, and (b) is distinguished by the fact that the junction output is not affected by its presence.

We use the value X*g* to represent the value carried by the output of a gate when its control is `1o`. X*g* represents a data output value that is unaffected by the data input. Consider two implementations of the gate primitive. It can be implemented as a 2-input AND or as a 2-input OR, as shown in Figure 15. In the AND implementation the inputs are A and B; the output is C. In the OR implementation the inputs are F and G; the output is H.

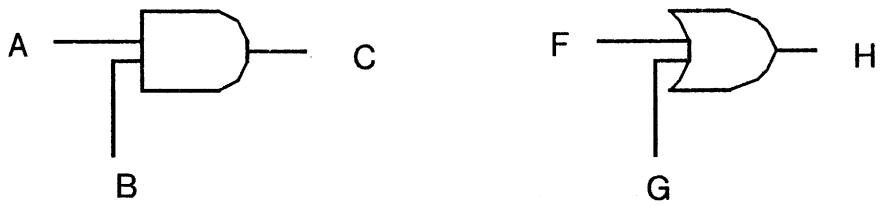
When the control B is 0, C is 0 and unaffected by A, so X*g* is 0. When the control G is 1, H is 1 and unaffected by F, so X*g* is 1.

The idea of a null value emerging from the gate is common to both implementations. We wish to capture this idea without worrying about whether the digital value should be 0 or 1. X*g* represents this abstraction. The example shows that X*g* (a) corresponds to some digital value on a wire, and (b) is distinguished because it represents a value insensitive to an input.

Finally, we use X*m* to represent the value carried by the output of a memory when the enable is `off` or the read/write is `write`. X*m* indicates that the output is not sensitive to the state.

Fig. 14. Two Implementations of the Junction Primitive

Fig. 15. Two Implementations of the Gate Primitive



3.3 Paths

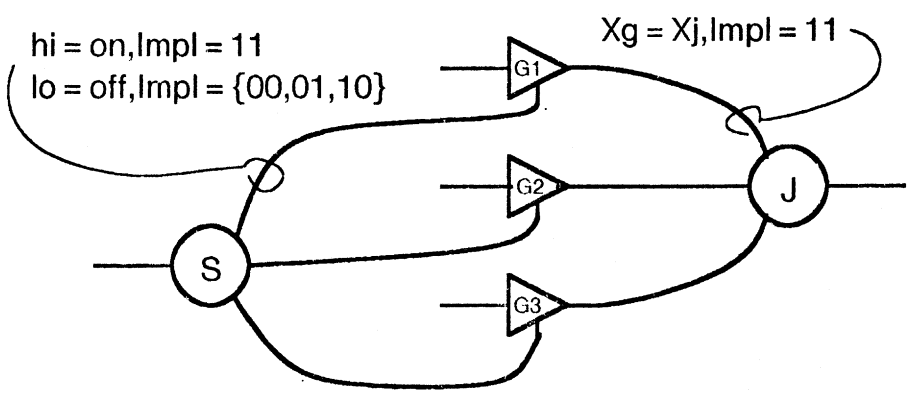
The paths that connect these functional primitives transmit any one of a set of values, for example, {hi,lo}, {on,off}, and {d*0*,d*1*...d*n-1*}.

Paths are annotated with several kinds of information. The two that will be important here are (1) design information about the intended interaction of the parts, represented as a mapping of input values to output values, and (2) information about the implementation technology, represented as a mapping from the information path values onto digital values.

The first mapping captures design information about the intended interaction of the devices. These intentions are represented by matching path input values to path output values. For example, part of the design specification for a multiplexer is that a lo input on a gate's control should ensure that the gate's data input will not affect the multiplexer output (Figure 16).

If the input of G1 is lo, it outputs X*g*. But the junction requires X*j* as an input. Thus the design of the multiplexer requires that X*g* and X*j* be equivalent; path DO1 is annotated with this information. Henceforth for simplicity we ignore the distinction between these two and use X to represent the common value.

Fig. 16. Design Information in the Multiplexer



Similarly, on paths E1, E2, and E3, the selector output hi maps onto gate control on, and selector output lo maps onto gate control off. This is more design information, this time about the intended interaction of the selector and gates. The mapping on E1, E2, and E3 allows us to use the values hi and lo henceforth rather than on and off.

The second kind of path annotation gives information about implementation. It shows how information path values map onto digital values. Consider for example the information path model representing a 4x2 boolean multiplexer (Figure 6). The control paths of the gates are each two bits wide; the digital value 11 maps onto hi, while the values {00,01,10} map onto lo. The gate output paths are each two bits wide, and the multiplexer output is insensitive to the values on these wires when they carry the digital value 11. That is, when the control input is lo, the outputs are 11. Thus in the boolean multiplexer X = 11.

3.4 The Fault Model

Our fault model deals with path behavior: a fault is always a fault in the transmission of values from one end of a path to the other. Restricting faults to appear only on paths maintains a useful level of generality that encompasses a wide range of physical faults.

Path behavior is described by a function E whose domain and range are the values that may be put on the path. Let d*i* represent a value transmitted by a path. If the path is not faulty, E is the identity function, so E(d*i*) = d*i* for all d*i*. If it is faulty, E(d*i*) = d*k*, where *k* is any 0..*n-1* except *i*. When a value d*i* is placed on the input end of a path, and the path has some fault E, the value that appears on the output end is E(d*i*).

For a faulty path, E models the behavior of a path which consistently transmits values incorrectly. Several inputs may map onto the same outputs, just as, for example, sa-0 maps the digits 0 and 1 onto 0. But because we are dealing with hard faults, E may not map a single input onto more than one output. For example, on a path with three values X, d*0*, and d*1*, the function { E(X) = d*0*, E(d*0*) = d*1*, E(d*1*) = d*1* } is a valid fault, since it obeys these restrictions.

Recall that D was defined in 3.2 to be the set of values {d0, d1..dn-1}. For a path whose range of values is the set D, E can describe all possible stuck-at and bridging faults in the underlying implementation. But for several reasons, this functional fault model is only an approximation of physical faults.

First, the fault model may not contain all the possible physical faults. Recall that in the boolean multiplexer, the gate control paths are two-bit paths whose range is {hi,lo}, implemented as hi = 11 and lo = {00,01,10}. Suppose the low-order bit of one of these paths is sa-1. When the path tries to transmit lo as 01 the fault will not be visible and E(lo) = lo. But when it tries to transmit lo as 10, the value 11 will appear, which is hi, so E(lo) = hi. Thus it has a sa-1 fault that violates the nonintermittency property of E.

Second, no fault in our model ever changes the direction of information flow, so that the "directionality" assumption is implicit. This is a common assumption to make when doing diagnosis, although it is sometimes violated in the real world [Davis 82].

Finally, faults in devices are not yet considered. The addition of such faults will increase the size of the problem, but we anticipate that it will not significantly change the design process. At this stage we feel that our fault model is a good enough approximation and general enough to be useful.

3.5 Inquiries

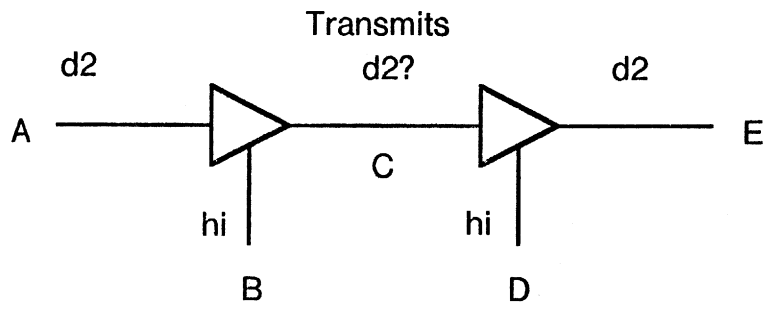
A fundamental concept in our work is the *inquiry*. An *inquiry* is a question of the form, "Does path P transmit the value *y* correctly?" The pair (P,*y*) is the *focus* of the inquiry. Each inquiry relies on some subset of paths within the device. Its reliance on those paths can be expressed as a set of *conditions* on those paths. If the inquiry fails, we conclude that one of the conditions was violated. Thus, the larger the set of conditions, the less specific the inquiry. Each inquiry consists of a single focus, a set of input values to the device, a test to be performed on its outputs, and some conditions. One of the conditions of the inquiry is that the focus is OK. Thus if the test fails we can infer that this condition may be violated and that the focus is not OK, i.e. that the path P fails to transmit *y*.

Figure 17 shows a simple example of an inquiry. It uses a simple device consisting of two primitive gates, and shows an inquiry about whether path C can transmit the value d2. Appendix I details the notation.

Values. To test whether C transmits d2, we must establish values at A, B, and D that cause d2 to be transmitted to E. Establishing d2 at A and `hi` at B causes C to transmit d2; establishing `hi` at D causes E to transmit d2. To determine the outcome of the inquiry we examine the output value at E.

Conditions. If C transmits d2 incorrectly, the inquiry will fail. Thus C appears in the conditions.[4] Likewise if B fails to transmit `hi` correctly, C will get the value `X` and the inquiry will fail. So B appears in the conditions. Similar arguments can be made for including A, D,

Fig. 17. An Inquiry




```
Inquiry I-1 : [ C ? d2 ]
Input Values: (A = d2) (B = hi) (D = hi)
Output Value: (E ? d2)
Conditions: (ok A) (ok B) (ok C) (ok D) (ok E)
```

4. The focus path of the inquiry always appears in the conditions, for the reason noted above.

and E in the conditions.

In general, conditions in inquiries arise because the inquiry depends on some subset of paths in the device. This subset typically includes paths that help establish an input on the focus (e.g. path B in Figure 17) and paths that help propagate the result to an output (e.g. path D). We will see that when devices with state are involved, the situation is slightly more complex: the inquiry will require a sequence of input values, and different paths may be involved at each step. The inquiry conditions will include every path relied on at any time during the sequence.

When an inquiry gets a bad result, we say that the inquiry *implicates* the paths mentioned in the conditions. Let a *test* be a series of inquiries, one for every value of a path.[5] If none of the inquiries implicates-- i.e., none of the inquiries detect a fault-- then we conclude that the focus path transmitted all its values correctly. In this case we say that the test *exonerates* the path.

We can now interpret the coverage and resolution criteria described earlier in light of the information path model. Diagnostics, which are collections of inquiries, must provide coverage and resolution. A diagnostic may be viewed as an attempt to exonerate all the parts in the device. Hence a diagnostic consists of an inquiry for every value on every path, to see whether the path can transmit the value faithfully. Exhaustive testing achieves coverage. To provide resolution, each inquiry should implicate as few paths as possible, that is, it should have the shortest possible list of conditions.

3.6 Conditions in the Information Path Model

A condition is an abstraction that captures a dependency between paths. In the previous example, to say that the inquiry for C includes the condition that B is OK expresses that dependency: it means that the inquiry for C depends on the behavior of B. But that statement loses information: it is stronger than necessary: B could be bad, but with E(hi) = hi and E(lo) = hi. In fact the inquiry shown earlier does not require that B is completely correct, only that it transmit hi correctly. To avoid overstating the case we need a vocabulary of conditions more expressive than simply OK.

In the forthcoming discussions we will use a hierarchy of conditions. Three progressively weaker conditions are used: OK, XOK, and XI. OK means that the path transmits all values faithfully. XOK means that it transmits at least the value X faithfully. The condition XI means "transmits information" and means that a path can transmit at least one bit of information. The hierarchy is a compromise that captures dependencies while allowing us to express conditions of varying strengths.

OK is the most common condition, although nearly always stronger than necessary, since in a given inquiry a path will typically need to transmit only one of its several possible values correctly. In most cases it doesn't hurt to make the conditions on each path too strong. This is because we are using the path conditions *only* to know what paths may have been

5. There are 2↑k inquiries per path in the the general case, but under a more restrictive fault model there are only O(k).

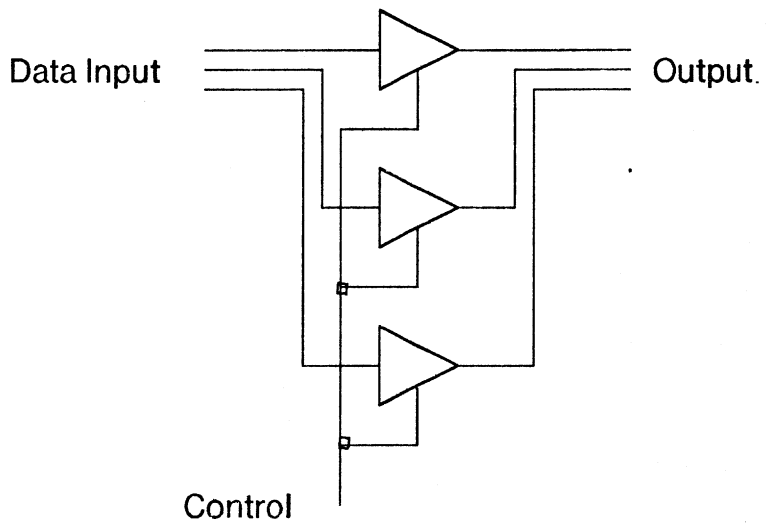
responsible for the inquiry failing: if the inquiry fails, we will know that one of the conditions has been violated; if the inquiry succeeds, we don't know anything except that the path under inquiry transmits that value correctly.

The condition XOK is motivated by the observation that many inquiries rely on ensuring that paths do not interfere with the propagation of results. Rather than use the condition OK, which suggests that a path strongly affects the inquiry result, we use XOK as the condition on those paths which should transmit X so as *not* to affect the result.

XI is the condition on a path whose information content need only be single bit. As an illustration of this idea, consider how one would test a gate device implemented as a group of tristates sharing a "control" input (Figure 18). We test the control by testing whether the output is sensitive to changes in the input. This is done by making the control 1 and examining the output when the input is 000, then examining it again when the input is 111. If the output changes we will conclude that the control is not sa-0. Likewise if we make the control input 0 and the output does *not* change when the input does change, we will conclude that the control is not sa-1. The test can be made robust against single-bit failures on the data input and output by examining the output with a "population count" rather than strict equality; that is, we can test whether the output has a majority of 1's or 0's. The resulting inquiry relies only on the ability to detect a change of majority on the output as a result of a change on the input. It thus requires that the input and output paths be able to transmit only one bit of information, i.e., two distinct symbols that represent these majorities.

The conditions from strongest to weakest are: OK, XOK, XI. XOK is weaker than OK because XOK can be satisfied even if a fault is present, while OK implies that no fault can be present. XI is weaker than XOK because a path that is XI may have E(d*i*) <> d*i* for all values of *i*, while a path that is XOK must have at least E(X) = X.

Fig. 18. Three Tristates implementing a Gate primitive



4. Overview of the Diagnostic Generation Procedure

Designing a diagnostic is done in three steps. First we create inquiries that cover all the paths in the device. From this step we get a set of inquiries, each with its own set of conditions. This is the **Inquiry Design** phase. Second, we analyze and combine the inquiries to reduce their conditions (thereby increasing their resolution) and improve their efficiency. This is the **Inquiry Improvement** phase. Third, we collect the reduced inquiries into tests and order the tests to take advantage of prior results. This is the **Test Ordering** phase. The resulting test sequence can then be translated into the actual diagnostic using implementation information.

4.1 The Inquiry Design Phase

During the Inquiry Design Phase we use an approach similar to path sensitization, applying it to our information-path model of the device. Recall from 2.1 that path sensitization works by *sensitizing* a fault and *propagating* the result. Henceforth, we will say that *backward* propagation of values sensitizes a fault, and *forward* propagation makes the result visible. To design inquiries, we propagate path values and path conditions. The local propagations are described by rules that propagate across single devices. There are currently 56 rules for the five primitives described in section 3.2.

There are four kinds of rules, capturing four different kinds of knowledge. *Behavior* rules describe the input/output behavior of devices. *Sensitization* rules capture knowledge about how values are propagated both forward and backward through devices. *Goal* rules guide the direction that the sensitization rules propagate. *Condition* rules capture knowledge about how the inquiries for a path depend on other paths.

A behavior rule is shown in Figure 19. This rule says that when the control of a gate is h i, the data input value is copied to the output.

Figure 20 is an example of a sensitization rule: to make the output of a gate d*i*, make the data input d*i* and the control input h i.

A goal rule is shown in Figure 21. This goal rule says that to observe a value on the control of a gate, values will need to be accomplished on the data input, and a value observed on the output.

Fig. 19. A Behavior Rule

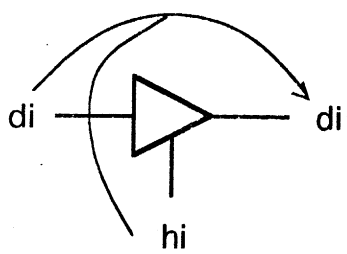


Fig. 20. A Sensitization Rule

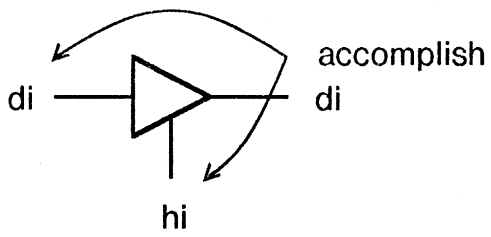


Fig. 21. A Goal Rule

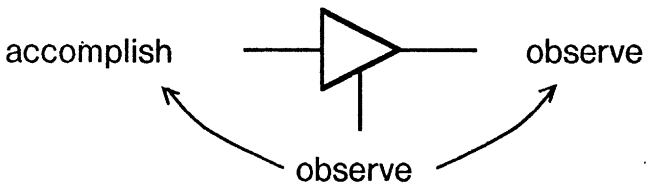
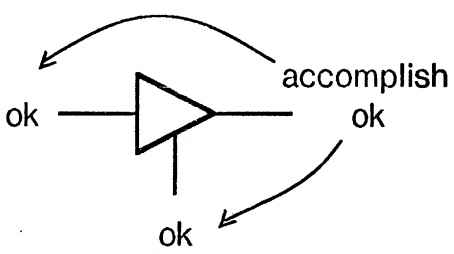


Figure 22 is an example of a condition rule: suppose the value on the output of a gate must be accomplished, and that the value must be OK. A fault on the data input could cause the output to appear bad, so the data input must be OK. Likewise a fault on the control input could cause the output to appear bad, so the control input must be OK.

To design an inquiry for a given path and value, we need to sensitize the path and propagate the result. To sensitize a path means we need to *accomplish* a particular value on it, and to propagate a result means we need to *observe* the result. Hence we begin the inquiry design process by annotating the focus path with the goals *accomplish* a value and *observe* it at an output. The rules then propagate goals, values, and conditions outward to the edges of the device.

For some rules, choices are available, and we iterate through these: upon reaching a contradictory assignment of values, we retract the most recent choice and go on to the next alternative. When the device contains elements with state, we stop propagating upon reaching state elements, and recursively perform the same procedure: once to accomplish the state inputs at an earlier time, and again to observe the state outputs at a later time. Thus inquiry design can be characterized as an exhaustive search for all input sequences that sensitize the focus and propagate its value. We will return to the search issue in section 8.4

Fig. 22. A Condition Rule



when discussing the system's implementation.

Consider designing an inquiry to see whether path DO-1 of the multiplexer transmits the value d2. We start by annotating the path DO-1 with the goals acc (accomplish) and obs (observe), the value d2, and the condition OK (Figure 23).

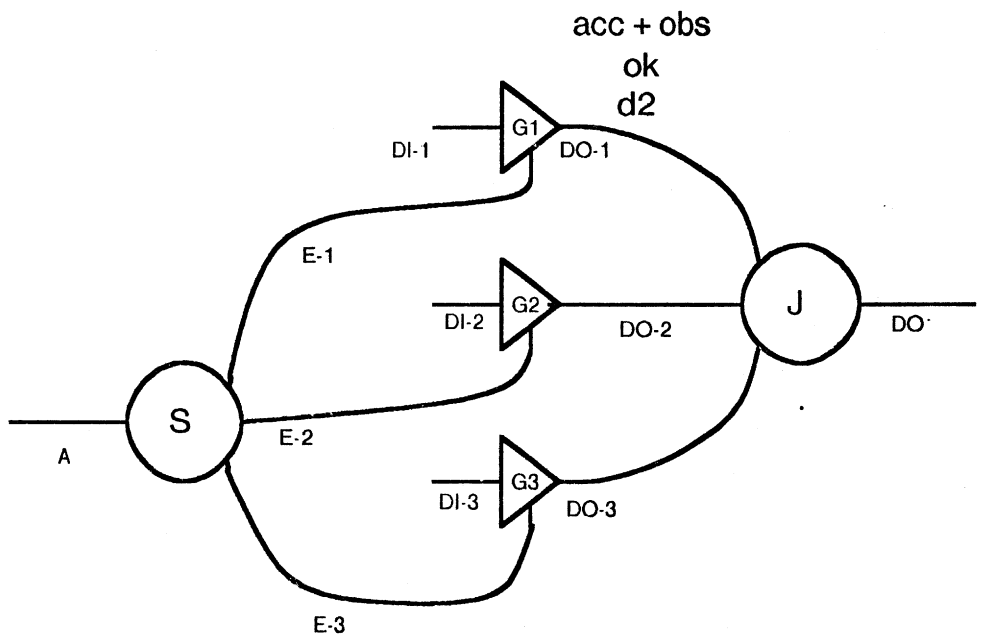
Rules fire to propagate goals, conditions, and values throughout the device; the final assignments are shown in Figure 24.[6]

The resulting inquiry is shown below.

```
Inquiry I-25 : [ DO-1 ? d2 ]
Inputs: (A = d1) (DI-1 = d2) (DI-2 = X) (DI-3 = X)
Output: (DO ? d2)
Conditions:
(ok A) (ok E-1) (ok DI-1) (ok DO-1) (ok DO-2) (ok DO-3) (ok DO)
(or (ok DI-2) (ok E-2)) (or (ok DI-3) (ok E-3))
```

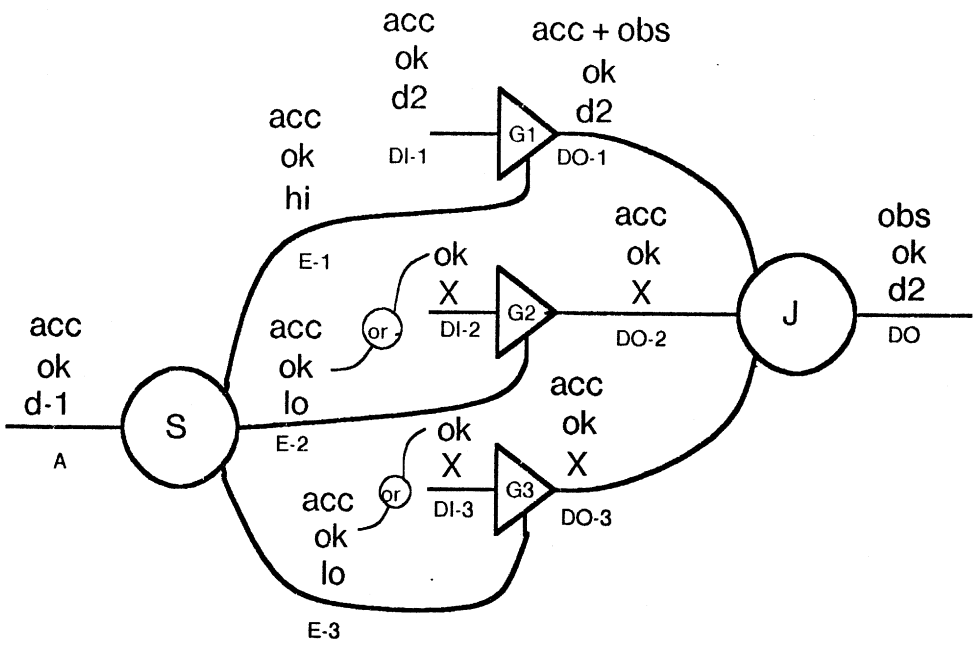
This inquiry means: To test whether DO-1 transmits d2, assign A to be d1, DI-1 to be d2, and let the other DI's be X. The test result can be observed by checking whether DO has the value d2. If the test succeeds, conclude that DO-1 can transmit d2. If the test fails, conclude that one of the paths A, E-1, DI-1, DO-1, DO-2, DO-3, or DO was bad, that E-2 and DI-2 were bad, or that E-3 and DI-3 were bad.

Fig. 23. Focus Assignments for the Inquiry [DO1 ? d2]



6. Around gates G2 and G3, the conditions on the control and data inputs have been or'd by a rule which checks whether X and 1o are present and if so "or's" the conditions.

Fig. 24. Multiplexer Assignments for the Inquiry [DO1 ? d2]



4.2 The Inquiry Improvement Phase

As we have noted, each inquiry relies on some set of paths, and an inquiry achieves better resolution as it relies on fewer paths. The Inquiry Improvement phase transforms each inquiry in an attempt to improve its resolution. This is done using the *single point of failure assumption*, hereafter SPFA. In our case, the SPFA is an assumption that only a single path is faulty; this is used to improve resolution in a number of ways. We describe each of these in turn. In addition to improving the inquiries' resolution, this phase also improves their efficiency by detecting and eliminating redundancies. We also use local implementation information to simplify inquiry conditions and further improve the inquiries' efficiency.

We can remove from inquiries all conditions which are guaranteed to be satisfied under the single point of failure assumption. During the construction of inquiries, it is sometimes necessary to create conditions of the form, "*either* A is ok *or* B is ok." Under the SPFA, such conditions are satisfied *a priori* and need not be considered further: if there is only a single fault, surely one of A or B must be ok. Thus we can simply drop such conditions from the inquiries.

Another technique used to reduce conditions is *collaboration*. Because choices are available during the inquiry design phase, we may find two inquiries for the same focus, each relying on different paths. Such inquiries can be combined into a single inquiry having a reduced set of conditions. We can do this under the SPFA because only conditions which appear in both inquiries could be responsible for both inquiries' failure.

Consider for example the situation in which we have two inquiries for A, one with the conditions (ok A) and (ok B), the other with the conditions (ok A) and (ok C). We can replace these with a single inquiry having only the condition (ok A): only a fault on A could

cause both original inquiries to fail: thus the new inquiry should have only A in its conditions. Collaboration combines inquiries with common foci; the new conditions are the intersection of the comprising inquiries.

In addition to these transformations which improve inquiries, we can improve the efficiency of the inquiries by *association*. If a set of inquiries with different values but the same focus rely on the same paths, their resolution is the same; all but one of the inquiries in these sets can be discarded. Similarly, inquiries with different foci may have the same inputs; only one need be performed.

Implementation information can be used to improve both resolution and efficiency. The *condition transformation* process finds inquiries in which paths that are several bits wide are used to transmit only a single bit of information. Under a special case of the single point of failure assumption, conditions on these paths are dropped. Certain test sequences can also be shortened by using this same information. This is described in detail in section 6.4.

4.3 Test Ordering

To exonerate a path under the current fault model, an inquiry must be performed for each of the values which could be transmitted by that path. If all such inquiries fail to implicate the path, then the path is exonerated. *Test aggregation* collects inquiries dealing with the same path to create a test for that path. For example, the inquiry which asks whether a gate control input can transmit `hi` and the inquiry which asks whether it can transmit `lo` comprise a test for that path.

The Test Ordering Phase then orders the resulting tests so that each has the minimal set of conditions. Tests' conditions can be reduced by ordering because any paths which have already passed their test are known to be good and hence need not appear in later tests' conditions. For example, suppose we have test T-1 with conditions `(ok A)` and `(ok B)` and test T-2 with the condition `(ok A)`. If we do the T-2 first, then T-1, the conditions in T-1 are effectively reduced to the single condition `(ok B)`.

4.4 Code Generation

We now use implementation information to translate the values in the inquiries into test vectors, and translate the inquiry conditions, removing any mention of previously exonerated paths, into "error messages."

5. Inquiry Design Rules and Their Use

The preceding overview of the three phases of the diagnostic generation process sketched the inquiry design phase. We now consider in some detail the rules that were used in that phase.

Recall that to design an inquiry for a given focus path and value, we use rules to propagate path values and conditions from the focus. As noted earlier, there are four kinds of rules, capturing four different kinds of knowledge: *Behavior*, *Sensitization*, *Goal*, and *Condition* rules. In the following sections we describe examples of these rules for the Gate, Selector, Junction, Fanout, and Memory devices. There are 56 rules in all.

We then illustrate an inquiry design for the multiplexer, and go on to describe how the notions of forward and backward propagation can be extended to design inquiries for devices with state.

5.1 Behavior Rules

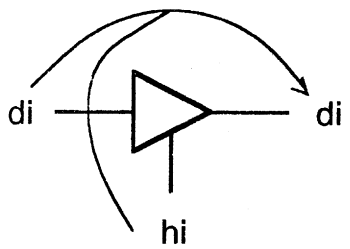
Behavior rules describe the behavior of devices.

The gate has two behavior rules: GB-1 (Figure 25) says that when the control is `hi`, the output gets the value of the data input; behavior rule GB-2 (Figure 25) says that when the control is `lo`, the output gets `X`.[7]

The selector has three behavior rules. Rule SB-1 says that the output of the *i*th output is `hi` when the address is d*i*. Figure 27 shows how the rule fires when the address is `d1`. The selectors' other two behavior rules allow us to infer from a `hi` output what the input must be, and from `lo` on *n*-1 outputs that the *n*th output must be `hi`.

The junction has two behavior rules. JB-1 says that the output is: `X` when all the inputs are `X`, d*i* when one input is d*i* and the rest are `X`, and undefined when more than one input is something other than `X`. Figure 28 illustrates case 2. A second rule infers from an `X` output that all inputs are `X`.

Fig. 25. Behavior Rule GB-1



7. In general, rules are written so that they require the fewest inputs, even though this may increase the number of rules. In this example, GB-1 requires that both gate inputs be known, while GB-2 requires only one.

Fig. 26. Behavior Rule GB-2

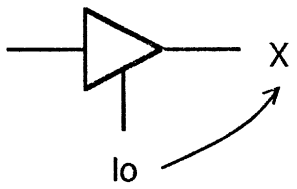
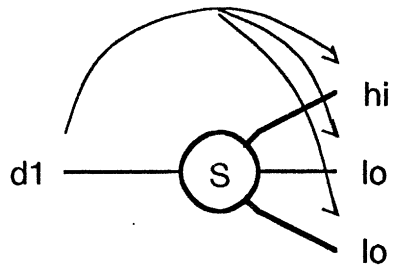


Fig. 27. Behavior Rule SB-1



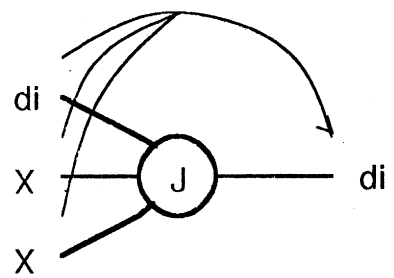
The fanout has two complementary behavior rules which express the idea that all the outputs must have the same value as the input, and that the input must have the same value as each output.

The memory has three behavior rules: one describes a write operation, one describes a read operation, and one describes its behavior when it is disabled.

5.2 Sensitization Rules

Sensitization rules are similar to the rules of traditional path sensitization and show (1) how information can be propagated forward through a device, and (2) how each possible output value can be achieved. The rules used in the *forward* direction can be expressed as, "to observe some input of a device, we need to accomplish some values on its other inputs and observe the output." The rules used in the *backward* direction can be expressed as, "to accomplish a value on the output of a device, we need to accomplish some values on the

Fig. 28. Behavior Rule JB-1



inputs."

An example of a forward propagating rule is GS-1 for gates (Figure 29): to observe any value on the data input of a gate, we must assign `hi` to the control input (if we assigned a `lo`, the output would always be `Xg`, that is, insensitive to the data input):

All the rules which propagate forward rely on the behavior of the devices and on the notion that when an input cannot be directly observed, we must accomplish values on its fellow inputs that will make that input visible.

An example of a backward propagating rule is GS-2 (Figure 30) for gates: to accomplish some value d*i* which is not equivalent to `X` (`Xg`) on the output, we need to accomplish that same value on the data input, and accomplish `hi` on the control.

The rules which do backward propagation, motivated by the need to accomplish a value on an output which cannot be directly set, nearly always involve choosing among the possible inputs which would result in that output. Rule JS-1 provides an example: to accomplish some value d*i* on the output, we must choose some input path on which to accomplish d*i*; `X` must be accomplished on the other inputs. Figure 31 shows the result of choosing the first junction input.

In Figure 31, exactly one of the junction inputs was chosen to transmit the value d*i*. For other devices the choices may be more involved. Rule GS-3 (Figure 32) provides an example: to accomplish the value `X` on the data output, we have a choice of two possible assignments, as shown in the two parts of rule GS-3. In the first choice, the control is `lo`. In the second choice, the data input is `X` (`Xg`). Note that we could use both `X` and `lo`. Thus they are not exclusive choices. We will return to this issue in section 8.4.

Fig. 29. Sensitization Rule GS-1

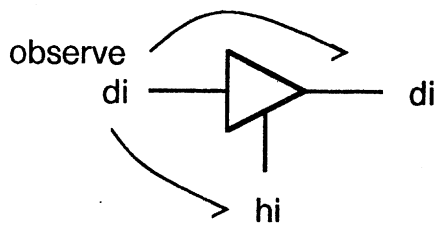


Fig. 30. Sensitization Rule GS-2

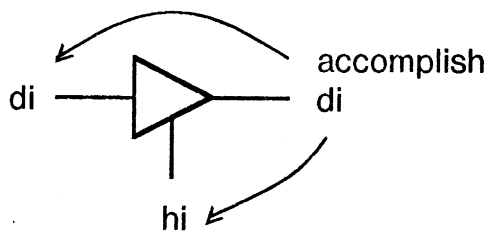


Fig. 31. Sensitization Rule JS-1

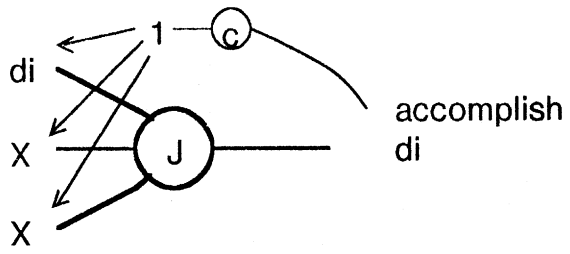
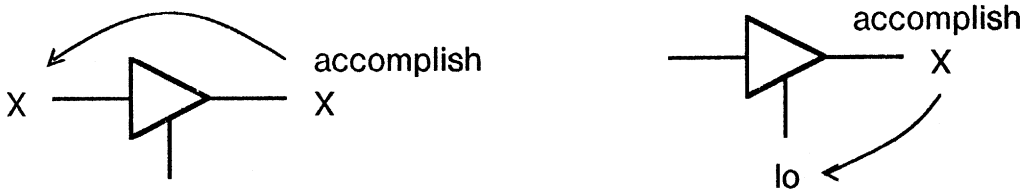


Fig. 32. Sensitization Rule GS-3



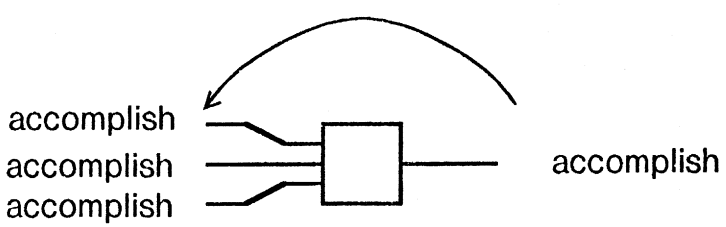
There are eight sensitization rules for the three devices used in the multiplexer; six more rules are used for the memory primitive.

5.3 Goal Rules

Goal rules tell which direction the sensitization rules will propagate, but do not assign values to the paths. Each path value must be either *accomplished*, meaning that backward propagation must occur from it, or *observed*, meaning that forward propagation must occur. Recall that backward propagation is guided by rules which can be expressed as, "if we wish to accomplish any value on the output, we need to accomplish some values on the inputs." Forward propagation is guided by rules which are expressed as, "if we wish to observe some input of a device, then we need to accomplish some values on its other inputs and observe an output. We call "observe" or "accomplish" the *goal* for a particular path. By assigning a goal of either *observe* or *accomplish* to a particular path the goal rules effectively set the direction of the sensitization rules.

Consider Figure 33, showing an arbitrary 3-input, 1-output device: to *accomplish* any value on the output, we need to *accomplish* some values on the inputs.

Fig. 33. Prototype Goal Rule for Backward Propagation



To *observe* some input of a device, we need to *accomplish* some values on its other inputs and *observe* the output (Figure 34).

To *observe* an input of a multiple-output device, we have as alternative observation paths any of the outputs of the device.[8] We conservatively assume that only one output will be observable. An example is shown in Figure 35. In general, any of the outputs may be chosen, although with some devices certain outputs will be unaffected by the input we wish to observe. Such outputs will not be valid alternatives.

There are seven goal rules for the three primitive devices used in the multiplexer; the fanout and memory primitives require eight more.

The focus of an inquiry has a path whose value must both be accomplished and observed. From this focus we can use the goal rules to assign a goal to each of the paths that will participate in the inquiry.

5.4 Condition Rules

Condition rules keep track of the interdependencies of establishing values on paths. These rules, like goal and sensitization rules, occur in both forward and backward varieties.

GC-1 (Figure 36) is an example of a backward condition propagation. To test a gate output to see whether it transmits a value d*i*, the control should be h i and data input d*i*. Since faults on either the control or data input would violate OK on the output, OK propagates to both

Fig. 34. Prototype Goal Rule for Forward Propagation

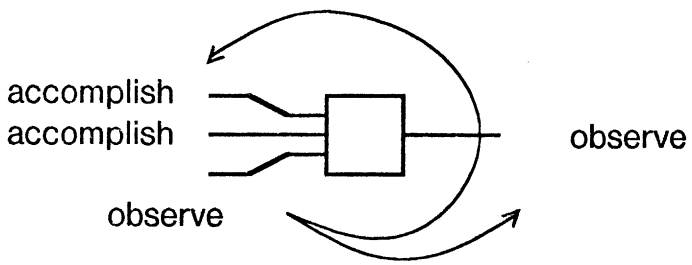
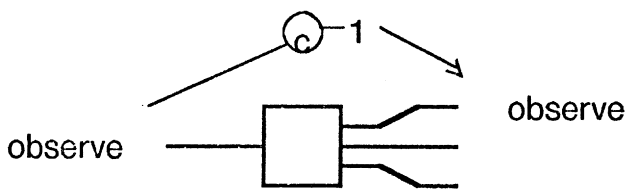


Fig. 35. Goal Rule for Forward Propagation with Multiple Outputs



8. Reconvergent fanout introduces a subtlety about backward propagation goal rules: rules for multiple-output devices must check first whether an input already has the goal "observe," since a single path may play both roles.

input paths.

This illustrates the basic principle behind condition rules: they add to the conditions of the inquiry wherever a fault might cause the same effect as violating an already existing condition.

GC-2 (Figure 37) is an example of a forward condition propagation. Suppose we wish to observe whether the control transmits the value `hi`; the condition on the control is `OK`. If we choose the data input randomly, we might end up with a value that could be mapped onto `X` by an error on the input or output paths. This would make it appear that the control was bad. Thus we have the condition that the input and output must be `OK`; that is, if the test fails, it could be because either of those paths was bad.

However, we can in fact get by with weaker conditions on the data input and output. Recall from 3.6 that the condition `XI` means "transmits information" and is satisfied by any path that can transmit one bit of information. We use `m0` to mean "a majority of 0's", and `m1` to mean "a majority of 1's." `{m0,m1}` is the set of values transmitted by a path that is `XI`. If we use one of the values in `{m0,m1}` on the data input and output, the only situation in which the control input could appear bad would be if the data input or output were incapable of transmitting even a single bit of information. Thus in this case we have the condition that the data input and output must be `XI`. The resulting rule GC-3 is shown in Figure 38.

Rule GC-4, shown in Figure 39, is an example of a rule that produces an "or" of conditions. Suppose we need to accomplish `X` on a gate output with the condition `XOK`. As we have seen in Sensitization Rule GS-3 (Figure -6), we can accomplish `X` by making the control `1o`, the data input `X`, or both.

Fig. 36. Condition Rule GC-1

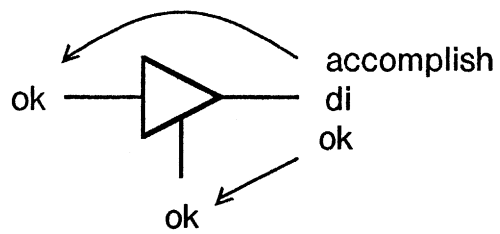


Fig. 37. Condition Rule GC-2

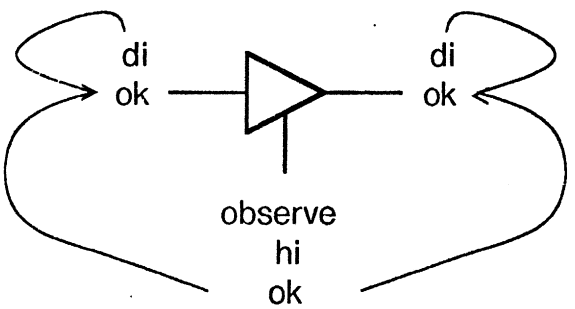
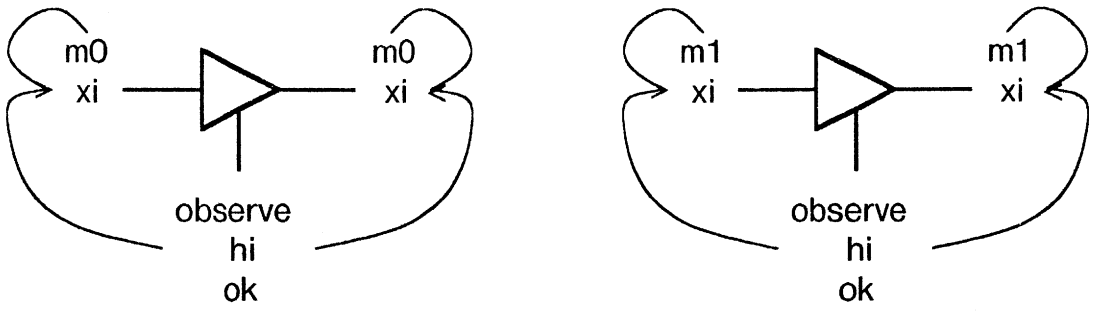


Fig. 38. Condition Rule GC-3



If lo is chosen as the control input then the condition on the control input is OK because an erroneous control value of hi could make the output appear bad.

If X is chosen as the data input then the condition on the data input is XOK because a bad data input could make the output appear bad.

But if both X and lo are chosen as inputs, then both inputs would have to be bad in order to make the output value bad. The proper condition is "either the data input is XOK, or the control input is OK." We can abbreviate this as (or (XOK data-input) (OK control)). The circled "or" notation indicates that a disjunct of conditions is present.

There are nine condition rules for the primitive devices in the multiplexer; there are seven more for the fanout and memory primitives.

5.5 An Example Of An Inquiry Design

The rules described above are used to design inquiries. We choose a focus path and value, annotate the path to indicate that we want to both "accomplish" and "observe" that value, and set the condition on the path to be OK. The rules fire and assign goals, values, and conditions to other paths in the device.

As an example we show the design of an inquiry for the multiplexer that tests whether path DO-1 can transmit the value d2. Figure 40 shows how the goal rules assign acc and obs to the paths in the multiplexer. Note that the address input A could be assigned acc from any of the selector outputs.

Figure 41 shows how the sensitization rules and device behavior rules use the previously assigned goals and the value d2 at DO-1 to assign values to the paths. For each of the gates G2 and G3, a consistent set of assignments is reached using behavior and sensitization rules. Both behavior rule SB-1 and sensitization rule GS-3 have determined the value lo for the control input. Both the behavior rule GB-2 and the sensitization rule JS-1 determine the value X for the data output.

Fig. 39. Condition Rule GC-4

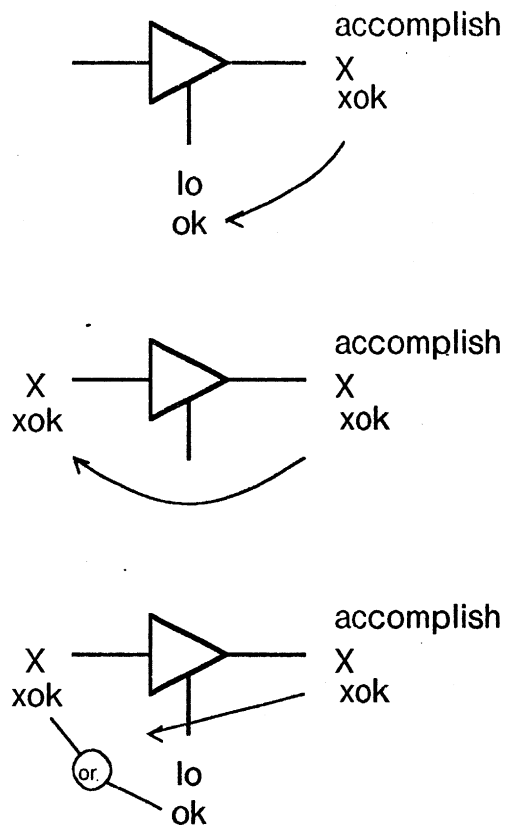
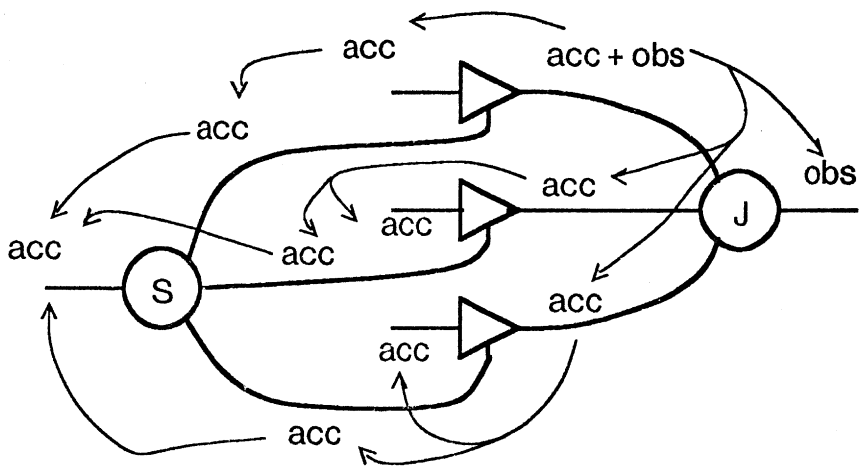


Fig. 40. Goal Assignments in the Multiplexer



The conditions on the multiplexer paths arise from the goals and values, and from the condition OK on path DO-1. Figure 42 shows the condition rules used and the final assignments.

Fig. 41. Value Assignments in the Multiplexer

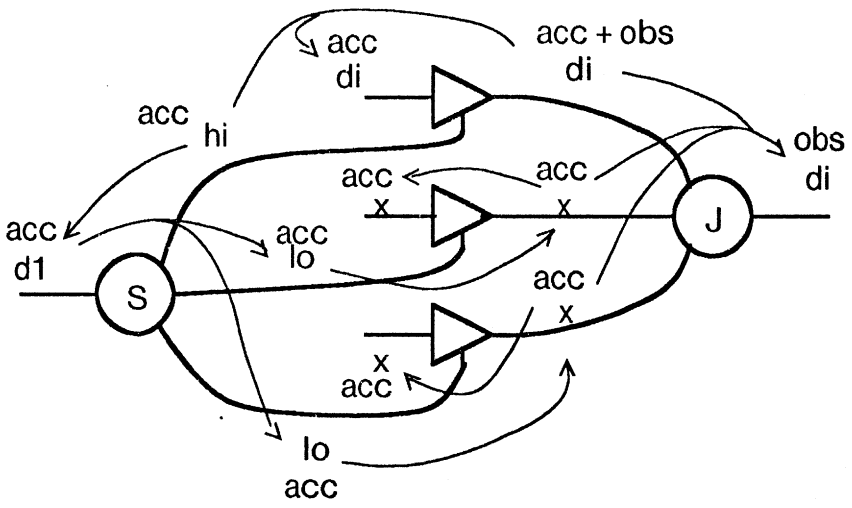
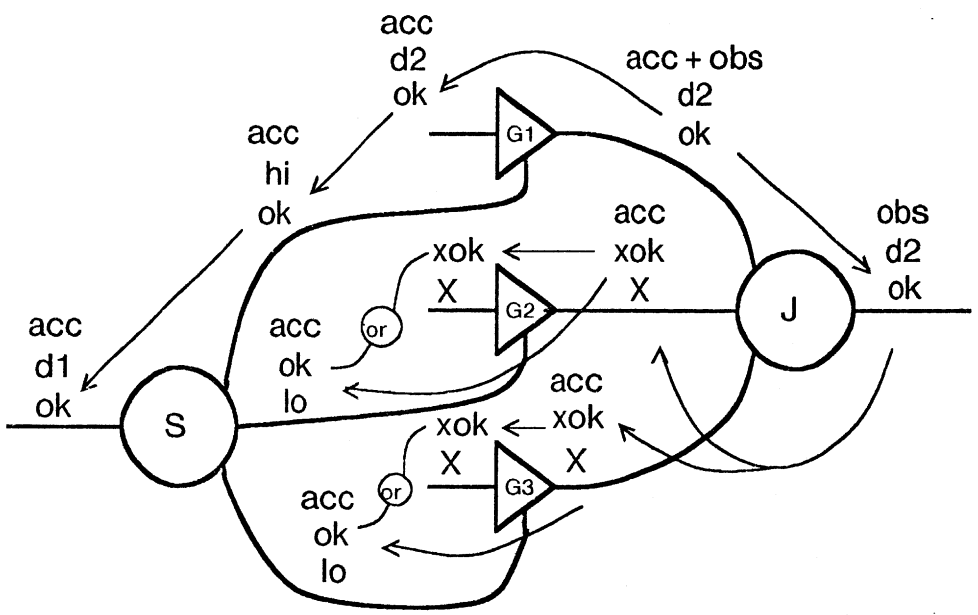


Fig. 42. Condition Assignments in the Multiplexer



Around Gates G2 and G3, condition rule GC-4 has been used, so that the conditions on the control and data inputs for each of G2 and G3 have been or'd. The propagation of the condition OK from the gate control paths E2 and E3 to A includes (ok A) as a condition in a disjunct of the or-clause, but is ignored. It is redundant, since it is already a member of the top level conjunct. The resulting inquiry is shown below.

```
Inquiry I-25 : [ DO-1 ? d2 ]
Inputs: (A = d1) (DI-1 = d2) (DI-2 = X) (DI-3 = X)
Output: (DO ? d2)
Conditions:
(ok A) (ok E-1) (ok DI-1) (ok DO-1) (xok DO-2) (xok DO-3) (ok DO)
(or (xok DI-2) (ok E-2)) (or (xok DI-3) (ok E-3))
```

This inquiry means that to test whether DO-1 transmits d2: assign A to be d1, DI-1 to be d2, and let the other DI's be X. The test result can be determined by checking whether DO has the value d2. If the test succeeds (i.e. d2 appears on DO), conclude that DO-1 can transmit d2. If the test fails, conclude that one of the paths A, E-1, DI-1, DO-1, DO-2, DO-3, or DO was bad, that E-2 and DI-2 were bad, or that E-3 and DI-3 were bad.

5.6 Propagation Through Time

Real computers contain memories of all sizes and types, so it is important for a diagnostic generation system to deal with state devices. Testing a device with state requires using a sequence of inputs, only at the end of which can the result be examined. For example, to test whether the output of a master-slave flip-flop is sa-0, we must write a 1 into the flip-flop on one clock tick and look at the output on the next. To design such a test, we would start with the goal of sensitizing the flip-flop output, and realize that to set it to 1, we must set the contents of the flip-flop to 1. Master-slave flip-flops only change their outputs after a clock tick, so we must write the 1 during a previous clock.

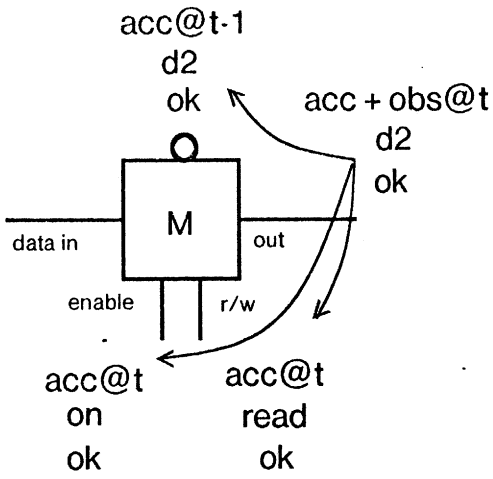
This kind of reasoning is quite analogous to the backward propagation that the system already does, but now we propagate back in *time* as well as *space*. To design inquiries for devices containing the memory primitive, we must propagate values and conditions forward in time to *observe*, and backward in time to *accomplish*. As before, reaching primary inputs and observable outputs terminates the propagations.

To illustrate propagation backward through time, we show the design of an inquiry that tests whether the output path OUT of the simple memory device in Figure 43 can transmit d2. We begin by annotating path OUT with the goal of accomplishing and observing the value d2 with the condition OK. The current time is *t*.

Recall from 3.2 that the the memory primitive has a state terminal whose value persists over time and is transmitted to the output when the R/W control is read. The state terminal can be annotated with a goal, value, and condition, just like an information path. A state terminal with the goal "accomplish at a previous time" is a *state input*; similarly, a state terminal with the goal "observe at a future time" is a *state output*. Whenever we have a state input, we must propagate backward in time to *t-1*. Similarly, we propagate forward from state outputs. Thus, the rules annotate state terminals during the current time, but only use those annotations for further propagation during a previous or future time.

Our system accomplishes this propagation by allowing the rules to run to quiescence and then taking a *snapshot*. A snapshot records the primary and state inputs, and the observable or state output during one clock tick (recall from section 5.3 that only one output is ever observed, so that there is never more than one primary or state output in a snapshot). The snapshot below is taken at *t* and corresponds to Figure 43. For simplicity of presentation,

Fig. 43. Propagations at Time t



conditions have been omitted from the snapshots.

```
Snapshot at Time t
Primary Inputs : (EN = on) (R/W = read)
State Inputs   : (STATE = d2)
Observable Output : (OUT ? d2)
```

This snapshot says that at time *t* we will perform a read operation on the memory and examine the output to see whether it is d2.

In order to get any value out of the memory at time *t*, we must get it onto the state terminal at time *t-1*, that is, we must accomplish d2 with the condition OK on the state terminal at *t-1*.[9] We make those assignments and get the results shown in Figure 44.

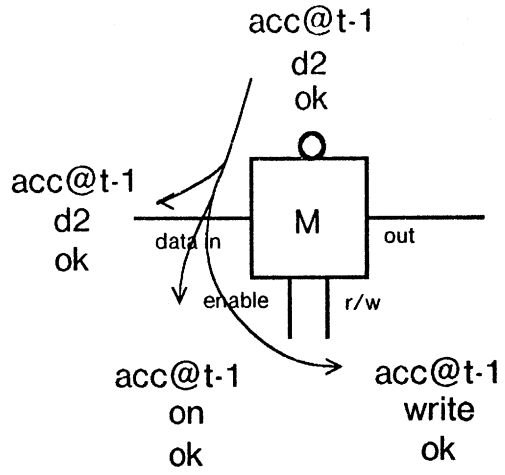
This results in the snapshot below. It shows that we will write d2 into the memory at *t-1*. There are no state inputs, so the system stops backward propagation. There were no state outputs at *t + 1*, so there are no forward propagations through time and we are done.

```
Snapshot at Time t-1
Primary Inputs: (IN = d2) (EN = on) (R/W = write)
State Output  : (STATE = d2)
```

The resulting inquiry is shown below and consists of the time-ordered sequence of the snapshots' primary inputs; its conditions are the union of the conditions accumulated at each time step.

9. Having taken a snapshot we can clear away the previous assignments. This allows paths to transmit different values during different time steps.

Fig. 44. Propagations at Time t-1



```
Inquiry I-37 : [OUT ? d2]
Inputs @t-1: (IN = d2) (EN = on) (R/W = write)
Inputs   @t: (EN = on) (R/W = read)
Output   @t: (OUT ? d2)
Conditions : (ok IN) (ok EN) (ok R/W) (ok OUT) (ok STATE)
```

This inquiry says that to test whether OUT transmits d2, write d2 into the memory at time *t-1*. At time *t*, read the memory and examine OUT to see whether it is d2. If it is not d2 then one of the paths IN, EN, R/W, or OUT, or the ability of the memory to store data, is bad.

5.7 Summary

We have seen how path sensitization can be used to design inquiries in the information path model. The connectivity and behavior of the device's components are used to drive the inquiry design process by propagating values backward and forward from the focus. This is done using sensitization rules; goal rules guide the sensitization rules to do forward or backward propagation. We have also seen that conditions can be propagated using rules similar to those for sensitization. Finally, we have seen that analogous "backward" and "forward" propagations can be used to design inquiries for devices containing state elements.

6. The Inquiry Improvement Phase

The inquiry improvement phase improves the resolution of the inquiries by reducing their conditions. This is done using a number of techniques, including the use of the *single point of failure assumption* to reduce the conditions in inquiries, and to combine inquiries so that their conditions are reduced.

The single point of failure assumption (SPFA) assumes that there is only one fault in the device. This is both a reasonable model of real-world faults and a useful simplification for test generation. The SPFA narrows considerably the conclusions that can be drawn from any given test: to explain any misbehavior, there are far more hypotheses involving multiple failures than those involving single failures. Thus, using the SPFA can be viewed as an application of Occam's razor.

The two uses of the SPFA in this phase are: elimination of "or" conditions, and *collaboration*, which combines inquiries. A specialization of the SPFA, the *independent channel assumption*, uses local implementation information to reduce conditions.

This phase also improves the efficiency of the inquiries by eliminating redundancies both among alternative inquiries for the same focus, and among similar inquiries with different foci. More local implementation information can be used to reduce the length of some test sequences.

6.1 Elimination of "OR" Conditions Using the SPFA

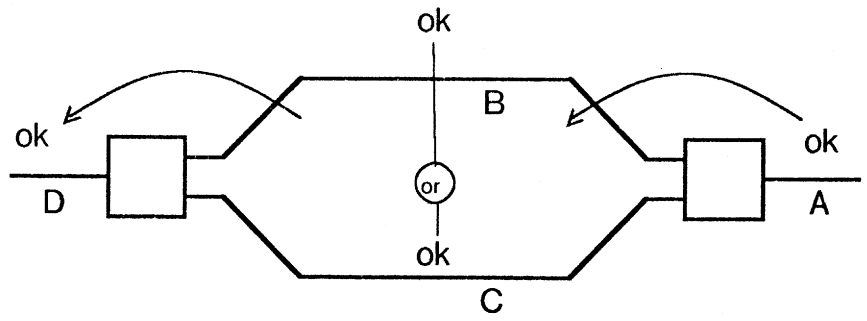
Some of the conditions created during inquiry design contained "or"'s, most commonly as `(or (ok control) (ok data-input))`. Under the SPFA we know that only one path, if any, is bad, so that such conditions can be satisfied immediately. In this case either the control or the data input must be OK. Here, we simply drop all such clauses from the conditions of our inquiries. Below, Inquiry I-19 is shown before and after or-elimination.

```
Inquiry I-19 : [ DI-1 ? di ]
Inputs: (A = d-1) (DI-1 = di) (DI-2 = X) (DI-3 = X)
Output: (DO ? di)
Conditions:
(ok A) (ok E-1) (ok DI-1) (ok DO-1) (xok DO-2) (xok DO-3) (ok DO)
(or (xok DI-2) (ok E-2)) (or (xok DI-3) (ok E-3))
```

```
Inquiry I-19 : [ DI-1 ? di ]
Values: (A = d-1) (DI-1 = di) (DI-2 = X) (DI-3 = X) (DO ? di)
Conditions:
(ok A) (ok E-1) (ok DI-1) (ok DO-1) (xok DO-2) (xok DO-3) (ok DO)
```

Since or'd conditions are going to be dropped from inquiries, it might appear that they should not be generated during inquiry design. This can be done, as long as fanout and reconvergence are taken into account. Consider Figure 45.

Fig. 45. Condition Resulting from Or-Conditions on Reconvergent Paths



D should be included in the conditions for any inquiry about A because if it is bad the values on *both* B and C will be bad. This has the same effect as violating `(or (ok B) (ok C))`.

6.2 Collaboration

Collaboration creates a new inquiry with reduced conditions by using evidence from at least two inquiries. Consider two inquiries that have the same focus but have different conditions, as shown below.

```
I-1
[ A ? d ]
Inputs: ...
Output: ...
Conditions: (ok A) (ok B)

I-2
[ A ? d ]
Inputs: ...
Output: ...
Conditions: (ok A) (ok C)
```

Suppose Inquiry-1 implicates. Then one of A, B, or C is bad. Suppose also that Inquiry-2 does *not* implicate. Now we cannot implicate A. Further inquiries would be necessary to distinguish between B and C.

Now suppose that Inquiry-1 and Inquiry-2 both implicate. Then one of A, B, or C is bad, AND one of A, B, or D is bad. Under the SPFA, C cannot be bad because this would not explain the result of Inquiry-2, and D cannot be bad because this would not explain the result of Inquiry-1. So, the conditions of the new inquiry I-3 are `(ok A) (ok B)`, because these are the only paths it can implicate. Inquiry-3 is shown below. It has two input/output sets corresponding to those of the comprising inquiries. Inquiry-1 and Inquiry-2 are no longer needed and can be deleted.

```
I-3
[A ? d]
Inputs 1: ...
Output 1: ...
Inputs 2: ...
Output 2: ...
Conditions: (ok A) (ok B)
```

The conditions of the new inquiry are the intersection of its constituent inquiries' conditions. Hence a compound inquiry is more robust and has better resolution because it achieves isolation from faults that violate conditions in only one of the two inquiries.

An interesting example of collaboration occurs among the three inquiries for [DO ? d*i*]. Inquiry I-19, shown below, is one of these three inquiries and transmits d*i* along the paths DI-1 and DO-1 to get to DO. Inquiries I-20 and I-21 are similar, but use different routes for d*i*.

```
Inquiry I-19 : [ DO ? di ]
Inputs: (A = d1) (DI-1 = d2) (DI-2 = X) (DI-3 = X)
Output: (DO ? d2)
Conditions:
(ok A) (ok E-1) (ok DI-1) (ok DO-1) (xok DO-2) (xok DO-3) (ok DO)
```

There is a subtlety about intersecting conditions: the intersection of (ok DO-1) and (xok DO-1) is the the condition (xok DO-1), since the condition that path DO-1 must transmit X correctly is subsumed by the condition OK.

Collaboration results in a single inquiry I-38 with reduced conditions. The comprising inquiries I-19, I-20, and I-21 are deleted.

```
Inquiry I-38 : [ DO ? di ]
Inputs 1: (A = d-1) (DI-1 = di) (DI-2 = X) (DI-3 = X)
Output 1: (DO ? di)
Inputs 2: (A = d-2) (DI-1 = X) (DI-2 = di) (DI-3 = X)
Output 2: (DO ? di)
Inputs 3: (A = d-3) (DI-1 = X) (DI-2 = X) (DI-3 = di)
Output 3: (DO ? di)
Conditions: (ok A) (xok DO-1) (xok DO-2) (xok DO-3) (ok DO)
```

6.3 Transforming Conditions Using Implementation Information

Another tool for reducing conditions is the mapping of information paths back onto the actual digital implementations. This mapping can be used to reduce or transform conditions.

The *independent channel assumption* is that all paths that transmit more than two values may be viewed as bundles of single-bit paths. With the SPFA we conclude that any multi-bit path is guaranteed to have at least one good single-bit path, and thus transmits at least two values even when faulty.[10] Recall that XI is the condition that a path need transmit only a single bit of information. For any path P that is more than one bit wide, the condition (xi P) is automatically satisfied under the independent channel assumption.

We can thus transform our existing inquiries by deleting XI conditions on paths that are more than one bit wide. In addition, XI is equivalent to OK for paths that are only a single bit wide, so we transform the inquiry to reflect this by changing XI conditions to OK.

6.4 Improving Inquiry Efficiency

Having reduced the conditions in the inquiries to achieve resolution, we can now eliminate redundancies to improve the inquiries' efficiency.

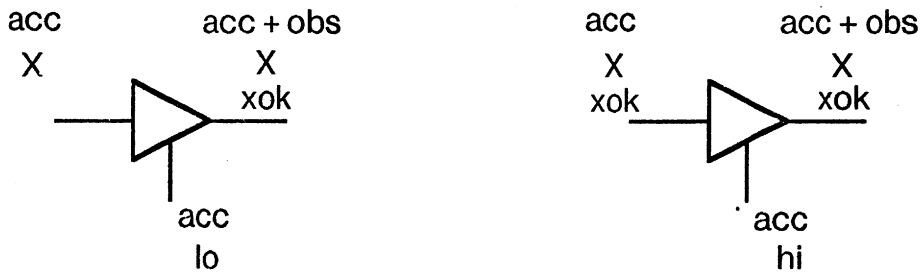
Recall that the existence of choices during inquiry design can lead to multiple inquiries for the same focus. For example, two inquiries could be designed for a simple device consisting of a single gate, as shown in Figure 46. As in this case, the multiple inquiries may have different inputs and the conditions of one may be a subset of another. Typically this occurs after or-conditions have been eliminated under the SPFA.

The conditions of I-1 are a subset of those of I-2, so I-1 has better resolution. In such cases we discard the inquiry with more conditions, retaining the inquiry with more resolution.

There are also cases in which inquiries for different foci have identical inputs. For example, an inquiry to see whether a gate's data input transmits d2 will be the same as one to see whether its output transmits d2. In such cases, we note that the inquiries are identical so that the resulting diagnostic will perform only one of them.

The efficiency of individual inquiries may also be improved by using implementation information. Recall from section 3.6 that we test whether a gate control path transmits hi by making the control hi and then examining the output to see whether it changes when the input changes. The example there showed that to test a gate implemented as a group of tristates, we could make the control input 1, change the data input from 000 to 111, and examine the output to see whether it too changed.

Fig. 46. Two Inquiries for a Simple Device




```
I-1: [Output ? x]                 |  I-2: [Output ? x]
Inputs: (Data = x) (Ctl = lo)     |  Inputs: (Data = x) (Control = hi)
Output: (Output ? x)              |  Output: (Output ? x)
Cond's: (xok Output)              |  Cond's: (xok Data) (xok Output)
```

---

10. This excludes from consideration such "single faults" as a disconnected bus cable that actually change the behavior of all the wires making up an information path.

A similar test could be written for the control input of a gate device implemented as a group of AND gates. However, the more general test can be made more efficient in this case by using implementation information. For the AND gate implementation we only need 111 as an input to tell whether the output is sensitive: if the output is closer to 000, we conclude that the output is not sensitive and thus the control is not transmitting 1; if the output is closer to 111, we conclude that it *is* sensitive and the control is indeed transmitting 1.

We can describe this in terms of the information path model. The output path of the gate implemented as tristates transmits the values {X,d0,d1...dn-1}. On the other hand, the output path of the gate implemented as AND-gates transmits the values {d0,d1...dn-1} and X, which is implemented as 000, happens to be identical to d*0*. In this latter case, inquiries that would require two input sets because they must transmit both elements of the set {m0,m1} are transformed to have only one input vector that uses the value d*n-1* to represent *m1*.

7. The Ordering Phase

The test ordering phase improves resolution by ordering the tests so that each has the minimal set of conditions. Tests' conditions can be reduced by ordering because any paths that have already been tested need not appear in later tests' conditions. In other words, if an inquiry relies on a previously exonerated path, that path appearing in its conditions may be ignored because the condition is known to be satisfied.

For example, assume that A has been exonerated previously, and the inquiry for B only has the conditions (ok B) (ok A) (ok C). If the inquiry for B indicates a bad result, only B or C could be bad; A is known to be good. This is a powerful technique that explains the usual tendency of hand-written diagnostics to test in a progressive way, beginning with those parts that are required for communication with parts to be tested later. A memory test, for example, would check the read/write enable control before moving on to test the data integrity.

The ordering is accomplished by comparing tests pairwise using three rules. We will show the rules and the ordering of the tests developed for the multiplexer. First we describe how tests are created from inquiries.

7.1 Test Aggregation

We defined a *test* for a path as a series of inquiries, one for each value on the path. When an inquiry has been run for every value of a particular path and none of the inquiries has implicated, then the focus path has transmitted all its values correctly. In this case we say that the test *exonerates* the path.

The general form of a test is shown below along with an example, the test created for E-1 from inquiries for the values hi and lo.

```
Test T-n : (ok ? <path>)
Inquiries: <list of inquiries>
Length: <number of value sets in all inquiries>
Conditions: (<cond> <path>) (<cond> <path>)


Test T-2 : (ok ? E-1)
Inquiries: I-10, I-43
Length   : 3
Conditions: (ok A) (ok E-1)
```

Test-2 uses three test vectors and was constructed from the inquiries I-10 and I-43. If neither of the inquiries implicate, we will conclude that E-1 is OK. If either implicates, then we look at the inquiries' conditions to see which of the paths are implicated in addition to E-1. The tests' conditions are the union of its constituent inquiries' conditions and are used as an ordering criterion.

A useful abbreviated notation for tests shows only the path under test and the conditions. Below is the abbreviated form of Test-2, whose focus is represented by (ok? E-1).

```
(ok? E-1) : (ok A) (ok E-1)
```

7.2 Rules for Pairwise Ordering of Tests

The basic principle behind ordering of tests is to perform them in the order that minimizes the conditions required at each step. We do this by ordering pairs of tests so that the tests with the weaker conditions are done first.

Conditions can be thought of as dependencies between tests: the test for path A depends on the test for path B if B appears in the conditions of the test on A. Thus for any group of tests we can build a directed graph whose links indicate "depends on." During this phase we want to change this directed graph into an acyclic directed graph and the resulting partial ordering into a total ordering. We can do this by collapsing loops into single nodes, eliminating redundant transitive links, and adding links where the ordering is left ambiguous.

There are two subtleties to this ordering. First, two paths may depend on each other. Since the resulting tests for those paths are not independent, they should be done close together in time to preserve the outward coherence of the resulting diagnostic. This is a matter of esthetics and has no effect on the logic of the tests.

Second, there are different strength links because the conditions have different strengths. The dependence of A on B resulting from (ok? A) : (xi B) is weaker than the dependence of B on A from (ok? B) : (ok A). Thus, while two paths may depend on each other, if A uses a weaker condition about B than B uses about A, we will test B first.[11]

With these foundations in place we now describe the rules. They are presented in both textual and graphic form. The text on the left shows a "before" and "after" transformation of test conditions. The graph on the right shows links representing the ordering on the nodes, which represent tests. Nodes nearer the top will be performed before those below. Unlinked nodes are unordered.

---

11. A related subtlety has to do with interpreting conditions. The test that went first because it had a weaker condition for some path P may be able to partially exonerate P, in particular when that weaker condition was XOK. A single inquiry is sufficient to determine whether or not (xok P): If the path is not XOK, then the condition would be violated and the inquiry would implicate. If the inquiry does not implicate, we conclude that all the paths with the condition XOK were properly transmitting X, and thus can delete XOK from the conditions of all the tests that follow. This is an exception to the notion that inquiries alone cannot exonerate paths. But we cannot similarly exonerate other paths mentioned with stronger conditions.

7.2.1 Ordering Rule 1: Postpone Tests with Strong Conditions

Postpone tests that could benefit from prior exoneration. Remove the satisfied conditions from the later tests.

```
(ok? Y) : ...
(ok? Z) : (ok Y) ...                                    (Y)
          |                                              |
          |                                              |
          V                                              |
Test Y before Z.                                        (Z)
(ok? Y) : ...
(ok? Z) : ...
```

Ordering Rule 1, case 1

This indicates that Z needs Y but Y does not need Z. Once the ordering has been established, the condition in the later test can be removed, because Y will already have been exonerated by `(ok? Y)`.

The same principle holds when Y *does* need Z, but requires only `(xok Z)` or `(xi Z)`: we still do Y first because it requires the weaker condition.

```
(ok? Y) : (<cond> Z) ...
(ok? Z) : (ok Y) ...                                    (Y)
                                                         |
==>                                                      |
                                                         |
Test Y before Z.                                        (Z)
(ok? Y) : (<cond> Z) ...
(ok? Z) : ...
```

Ordering Rule 1, case 2

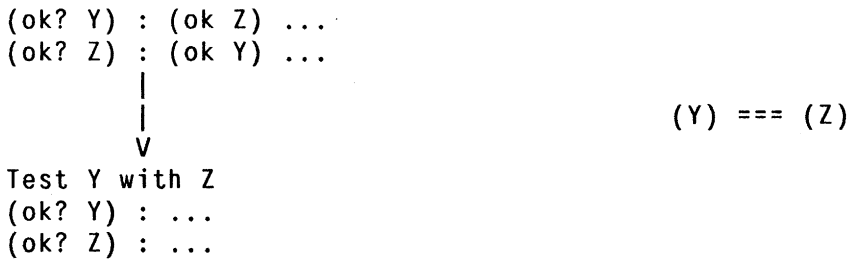
Ordering rule 1 orders multiplexer tests T-1 and T-11:

```
(ok? A)  :
(ok? DO) : (ok A) (xok DO-1) (xok DO-2) (xok DO-3)          (A)
           |                                                 |
           |                                                 |
           V                                                 |
Test DO after A.                                           (DO)
(ok? A)  :
(ok? DO) : (xok DO-1) (xok DO-2) (xok DO-3)
```

Example of Ordering Rule 1 Being Applied

7.2.2 Ordering Rule 2: Group Tests That Rely On Each Other

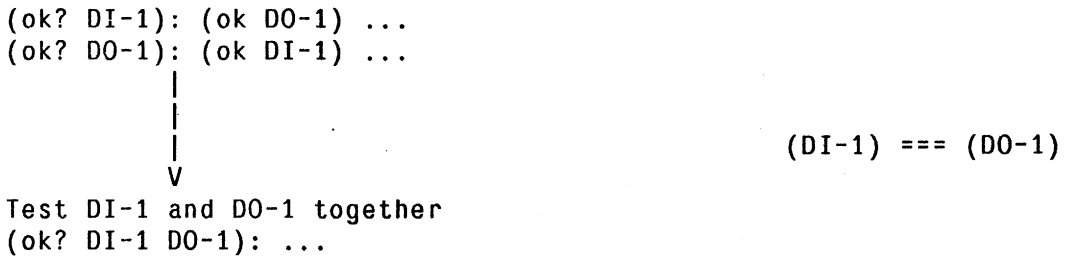
Some tests have each others' focus path in their conditions. They generally share inquiries. These tests should be ordered to be close together in time to give the diagnostic coherence.



Ordering Rule 2

This indicates that there is an overlap of implications caused by faults on Y and Z. In other words, during testing of Y, there are some inquiry results that could implicate either Y or Z, and likewise, during testing of Z there are inquiry results that could implicate Z or Y. It is possible that there are results that could only implicate Y, or only implicate Z, but since the tests generally will be sharing inquiries, they might as well be tested together.

An example of ordering rule 2 being applied between the tests for DI-1 and DO-1 is shown below.



Example of Ordering Rule 2 being Applied

7.2.3 Ordering Rule 3: Disjoint Conditions

The preceding two rules will not apply in all cases, because sometimes test conditions are disjoint. In this situation we do the tests with weaker conditions first. If the conditions cannot be characterized by relative strength, we order by doing first those tests that require fewer input sets in their inquiries. If the input sets are of equal length then the ordering is arbitrary.

In the multiplexer, the tests for E-1, E-2, and E-3 are disjoint from the conditions for DO. We apply this third rule to order the E-j's, which each require 3 value sets, before DO, which requires $3n + 3$ value sets.

7.3 Ordering the Multiplexer Tests

The set of tests developed previously is shown in Figure 47. For simplicity of presentation the XI conditions have been removed under the independent channel assumption, as described in section 6.3. They do not change the ordering.

Ordering rule 1 applies:
* Between (ok? A) and all the other tests
* Between each (ok? E-j) & (ok? DI-j) and between (ok? E-j) & (ok? DO-j)
* Between (ok? DO) and each (ok? DI-j) and (ok? DO-j).

Ordering Rule 2 applies between (ok? DI-1) & (ok? DO-1), between (ok? DI-2) & (ok? DO-2), and between (ok? DI-3) & (ok? DO-3).

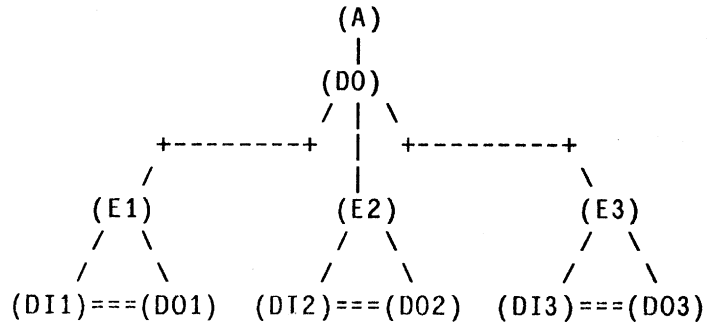
Ordering Rule 3 applies, as mentioned before, between each of the (ok? E-j) and (ok? DO).

This gives us the final ordering shown in Figure 48 and leaves the tests for E-1, E-2, and E-3, as well as the DI/DO-j pairs unordered.

Fig. 47. Multiplexer Tests with XI-conditions Removed

```
(ok? DO3) : (ok A) (ok DI3) (xok DO1) (xok DO2) (ok DO) (ok E3)
(ok? E3)  : (ok A)
(ok? DI3) : (ok A) (xok DO1) (xok DO2) (ok DO3) (ok DO) (ok E3)
(ok? DO2) : (ok A) (ok DI2) (xok DO1) (xok DO3) (ok DO) (ok E2)
(ok? E2)  : (ok A)
(ok? DI2) : (ok A) (xok DO1) (ok DO2) (xok DO3) (ok DO) (ok E2)
(ok? DO1) : (ok A) (ok DI1) (xok DO2) (xok DO3) (ok DO) (ok E1)
(ok? E1)  : (ok A)
(ok? DI1) : (ok A) (ok DO1) (xok DO2) (xok DO3) (ok DO) (ok E1)
(ok? DO)  : (ok A) (xok DO1) (xok DO2) (xok DO3)
(ok? A)   :
```

Fig. 48. Ordering of Multiplexer Tests after Rules 1, 2 and 3



This gives us the following sequence of tests:

(2) Test A : (ok? A) : (ok A)

This means that the test will either implicate A or exonerate A alone.

(4) Test E-1, E-2, and E-3 :

```
(ok? E-1) : (ok E-1)
(ok? E-2) : (ok E-2)
(ok? E-3) : (ok E-3)
```

This means that the tests for each E-j will will either implicate or exonerate that E-j alone.

(6) Test DO : (ok? DO) : (ok DO) (xok DO-1) (xok DO-2) (xok DO-3)

This means that the test can be written so that it can exonerate DO, and it can implicate DO, DO-1, DO-2, or DO-3.[12]

(8) Test each of DI-1/DO-1, DI-2/DO-2, and DI-3/DO-3 together

```
(ok? DI-1 DO-1) : (ok DI-1) (ok DO-1)
(ok? DI-1 DO-1) : (ok DI-1) (ok DO-1)
(ok? DI-1 DO-1) : (ok DI-1) (ok DO-1)
```

This means that for each j, a test can be written that will exonerate DO-j and DI-j, or will implicate DI-j or DO-j.

7.4 Local Preordering

The time needed to check for intersections between large sets of conditions degrades the performance of the ordering phase. We can improve this by using local information to decide which pairs of tests to try to order first. The heuristic is that pairs connected to a common device can be locally ordered. For example, consider a gate with data input IN, control CTL, and output OUT. Designing tests for these paths results in the tests shown below.

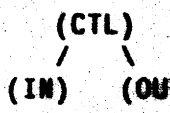
```
(ok? CTL) : (xi IN) (xi OUT)
(ok? IN)  : (ok OUT) (ok CTL)
(ok? OUT) : (ok IN) (ok CTL)
```

Between CTL and IN, and between CTL and OUT, we can apply Ordering Rule 1. Between IN and OUT, Ordering Rule 2 applies. Figure 49 shows the result of the ordering rules.

---

12. As described in the previous footnote, the condition XOK on the DO-k's means that if this test succeeds, then future inquiries with the condition (xok DO-k) cannot implicate DO-k.

Fig. 49. Local Application Of Ordering Rules to a Gate




```
(ok? CTL) : (ok CTL) (xi IN) (xi OUT)
(ok? IN)  : (ok IN)
(ok? OUT) : (ok OUT)
```

We can start by ordering the tests for the gate device, and then apply the other rules to the remaining tests.

8. Implementation

An implementation of the system has been written in Franz Lisp on a VAX-11/780 running Unix. Devices built from the primitives of the information path model are represented in a language described in [Davis 82]. In the inquiry design phase, the rules previously described are used to derive the consequences of goal, value, and condition assignments to paths. Because some rules require choices to be made, the program designs all the inquiries for a focus using exhaustive search. The search can be limited in several ways, as described below. Inquiry improvement and test ordering are programmed for the most part as straightforward set operations and sorting algorithms.

8.1 Rules

The description language, based on DPL [Batali 81], has a mechanism for implementing rules. Each device has some terminals; for example, the terminals of a gate device are the data input, the control, and the output. These terminals have *cells* that can be assigned values. Each rule has some input cells, a procedure, and an output cell where the procedure's value will be assigned. Each rule is "threaded" across cells at the terminals of a particular device. Whenever a cell is assigned a value, the list of rules that use the cell are checked to see if they can run, and those that can will run and cause other assignments to cells.

The propagation mechanism, which is based on the notion of constraints [Steele 80], causes the system to run rules and assign cells until quiescence is reached. Dependencies are maintained for cell assignments, so that retracting a cell upon whose value other cells depend will cause those other cells' values to be retracted as well. When terminals of devices are attached by an information path, their cells are shared, so that setting a value in a cell by one rule will cause the connected devices' rules to fire.

The inquiry design rules shown earlier are encoded as rules using this machinery. Every terminal has "goal," "value," and "condition" cells, across which the rules are threaded. Rules have the form (<output-cell> (<input-cells>) <lisp-form>). Rules run when all their input-cells are assigned. A rule may run and either fail or succeed. The form (whenever <condition> <lisp-form>) causes the rule to fail unless <condition> is true.[13] If the rule runs and succeeds, the output-cell is assigned the value that the lisp-form returns. Below are encodings of Behavior Rule GB-1 and GB-2.

```
(output-value (input-value control-value)
              (whenever (eq control-value 'HI) input-value))

(output-value (control-value)
              (whenever (eq control-value 'LO) 'X))
```

13. This is in contrast to (if <condition> <form>), which would cause the rule to succeed and return the value "nil".

As it happens, the rules can have only a single output cell, so a rule such as GS-2 must be encoded as two rules:

```
(input-value (output-goal output-value)
             (whenever (and (eq output-goal 'ACCOMPLISH)
                            (neq output-value 'X))
                       output-value))

(control-value (output-goal output-value)
               (whenever (and (eq output-goal 'ACCOMPLISH)
                              (neq output-value 'X))
                         !HI))
```

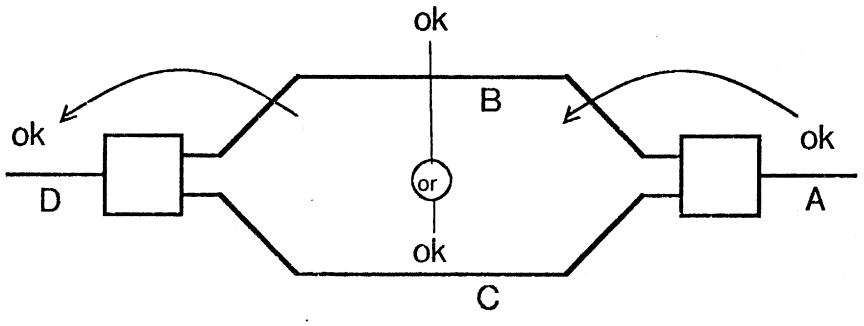
8.2 Avoiding Or-Conditions

Recall from section 6.1 that "or-conditions" result when a condition depends on any one of several conditions being satisfied and that these or-conditions can be eliminated by using the single point of failure assumption. Our implementation of the system avoids recording these conditions, thus bypassing or-elimination during inquiry improvement.

For example, in Figure 50, reproduced from section 6.1, the condition `(ok A)` results in the or-condition `(or (ok B) (ok C))`. `(ok D)` results from both `(ok B)` and `(ok C)`. Thus, the conditions will be `(ok A)`, `(ok D)`, and `(or (ok B) (ok C))`. The last of these conditions can be eliminated under the SPFA.

The program eliminates or-conditions during the inquiry design phase by making use of information maintained by the rule-running mechanism.[14] For any assignment to a path, such as `(ok A)`, the mechanism maintains all the consequences of that assignment. In the example above, `(ok D)` is a consequence of `(ok B)` and `(ok D)` is a consequence of `(ok C)`. Each condition in an or-clause has a set of consequences; the program only records conditions in the intersection of these sets. In the example above, only `(ok D)` would be recorded.

Fig. 50. Conditions Resulting from Reconvergent Paths



14. When state devices are involved, this information can be preserved in snapshots.

8.3 Avoiding Inquiry Designs

As noted before, the inquiry improvement phase detects and deletes redundant inquiries. Such inquiries can often be detected before the design begins by doing *fault collapsing*. For example, the fault E(d2)<>d2 on a gate input and E(d2)<>d2 on its output are indistinguishable, and so can be collapsed. Inquiries that test paths for indistinguishable faults will always turn out to be identical. Only one of each such pair needs to be generated.

We can also use *parameterization* to reduce the number of inquiry designs. For example, inquiries that test for d2 and d3 on the same path will nearly always be similar. We design such inquiries by propagating the symbolic parameter d*i* to represent the values d*0* through d*n-1* on the focus path.

8.4 Inquiry Design as a Search Problem

An assignment of a value and condition to a focus path results in the creation of choice points. To make all the inquiries for a particular focus, an exhaustive search must be made of all the choices. In general the size of the search tree has an upper bound of $O((n*m)\uparrow h)$, where h is the length of the longest path from an input to an output, n is the number of devices in each stage, and m is the number of possible choices for each device. Upon reaching a choice, the program creates a choice node and iterates through the possible assignments. This results in a depth first search.

Search may be avoided by making choices that are likely to yield better inquiries. Since we prefer inquiries with fewer conditions, we may search the tree in such a way that we try only those assignments that keep local conditions to a minimum. Pruning branches in this way has the effect of reducing the work done in the inquiry improvement phase. Recall from 6.4 that if two inquiries test the same path, we delete the inquiry whose conditions were a superset of the other's. Instead of deleting the redundant inquiry, we can avoid generating it in the first place. As with any search heuristic that prunes branches we must accept the possibility that we will miss winning branches, so we may in rare cases miss better inquiry designs.

An example of this heuristic is currently encoded into the program's rules. By comparing condition rules GC-2 and GC-3, it should be clear that observing whether the control input is `hi` or `lo` should be done using the inputs `{m0,m1}`, since this results in the condition `XI` on the output path, while any other data input assignment results in the condition `OK`, which is stronger. This cuts down the number of possible assignments from *n-1* to 1.

This heuristic also treats an "or" of conditions as weaker than, and thus preferable to, any of `XI`, `XOK`, or `OK` alone. This is obviously true under the SPFA, but still true even if we only assume that multiple failures are less likely than single failures.

The gate device provides an example of an "or" condition. Recall sensitization rule GS-4: to accomplish `X` on the output of a gate, we may either accomplish `X` on the data input, or accomplish `lo` on the control input. These choices typically result in the conditions `(ok control)` and `(ok data-input)`.

In reality there is another alternative: we can also assign *both* `X` and `lo` as inputs to the gate. The resulting conditions include `(or (ok control) (ok data-input))`, which may be dropped during or-elimination. Thus choosing both may yield an inquiry whose conditions are less than the conditions resulting from either alone.

The program takes advantage of this. We call the assignments of `X` and `lo` *nonexclusive* assignments. Upon reaching a set of nonexclusive assignments, it creates a choice node with three alternatives. The first alternative is to try making *all* the assignments; this terminates early and successfully if any inquiries are found. If that fails then it goes on to iterate through all the individual assignments.[15]

More than one such set of alternatives may be available at any point during an inquiry design. The order in which the sets are explored may prevent some inquiries from being found. For example, the inquiry `[DO ? x]` creates three choice nodes, at G1, G2, and G3. If we begin at the first choice node by assigning `X` and `lo` to G1's inputs, then making subsequent choices at G2 and G3, we find two inquiries with `(A = d2)` and `(A = d3)`. But we would miss the inquiry with `(A = d1)`. Because the order of exploration matters, the program tries all permutations of that order and avoids redundant sets of assignments by memoization.

15. The gate is a simple, 2-assignment case. The technique generalizes to trying all the subsets of the n assignments-- first the subsets with n elements, then the subsets with n-1 elements, etc.-- and allowing successful termination after any group of subsets. As a result an exponential number of subsets may be tried.

9. Future Directions

There is much more to the problem of designing diagnostics than covered by the current system. Some limitations become clear in examining the inquiry design methodology, the multiplexer problem on which it currently works, and its performance on a device containing state elements.

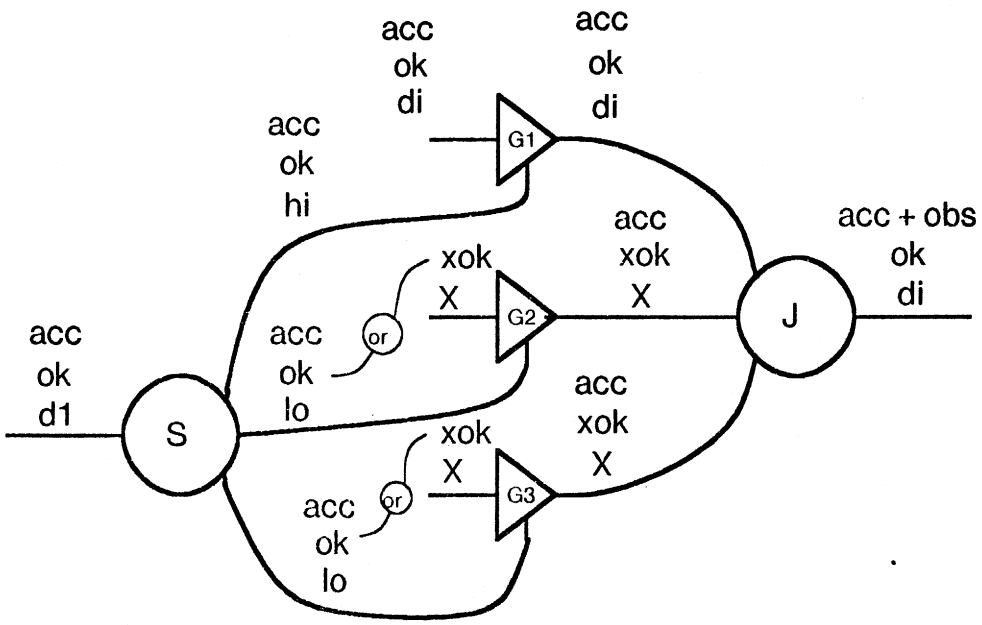
9.1 Limitations of the Inquiry Design Methodology

There are two important limitations of our current system that arise because the system is based on local propagations. First, the rules can only propagate single values, when at times sets of values would be more appropriate. Second, the system also needs nonlocal information about relationships between values on related paths. In a junction, for example, we may need to know that to obtain a d*i* at the output, exactly one of the inputs must be d*i* while the other inputs have X. Unfortunately such assertions about the behavior of the junction cannot be represented by local assignments to individual paths.

Consider an inquiry to see whether path DO of the multiplexer can transmit d*i*. Strictly applying the method previously described, we arrive at a set of inquiries differentiated only by the choice of path. So we would have separate inquiries that use gates G1, G2, and G3 to try to transmit the value d*i* from a primary input to the observable output. Figure 51 shows the value and condition assignments for one of the three inquiries.

All three inquiries have the condition OK on the address input, since a fault on this input would cause an incorrect data input to be selected.

Fig. 51. One of Three Inquiries for [DO ? di]



But if we put the value d*i* on all the data inputs at once, the choice of address input would be a don't-care, so that there would *not* be any condition on the address input. This is desirable because it reduces the conditions for inquiries about DO. To perform this task the system needs two important additions to its knowledge: paths transmit sets of values, and values on paths may be related in ways not captured by device behavior rules. The system also needs an addition to its notion of goals.

9.1.1 Paths Transmit Sets of Values

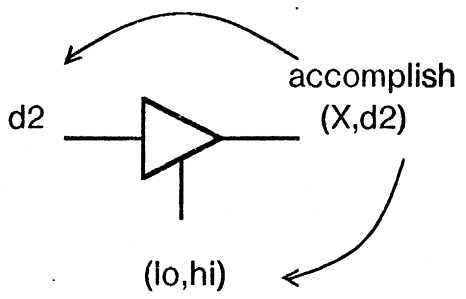
The rules currently do not capture the knowledge that a properly behaving device may constrain its paths to have a range of values. For example, to accomplish the value d*i* on the output of a junction, each individual input will be either d*i* or X. But the rule can only make a choice among those values, which is not the same thing.

We can use such information to constrain the choice of gate inputs. Because each input of the junction is either d*i* or X, each gate output is either d*i* or X. For any input value of the selector, every output will be either hi or lo; from this we see that for each gate data input is either hi or lo. Finally, we know that when the gate's input is d*i*, its output is either d*i* or X depending on whether its control is hi or lo.

We can achieve this effect by propagating sets of values. For example, DO-1 may be assigned the value "(X, d2)", meaning "either X or d2." We can make a sensitization rule using the knowledge that when the data input is d*i*, the output is either X or d*i*. The sensitization rule shown in Figure 52 says: to accomplish (X, d*i*) assign the data input to be d*i* and assign the control input (hi, lo).

Unfortunately such a propagation is unmotivated if all we know is that the output *could be* one of X or d*i*: we have no reason to choose d*i* as an input unless there is some case in which the output *must* be d*i*.

Fig. 52. Sensitization Rule GS-5



9.1.2 Global Information Is Available

We need information about the relationships between values on related paths. For the junction in the example above, we know that exactly one of the inputs must be d*i*. For the selector, we know that exactly one of the outputs must be hi. Thus we can choose d*i* for every gate's input value to cover the cases when the control is hi and when it is lo. Choosing the input on every gate to be d*i* allows us to "match up the cases" on both the input and output.

Unfortunately such assertions as these about the behavior of the selector and junction cannot be represented by local assignments to individual paths. This problem occurs in virtually any system relying on local propagation.

One answer to this problem is to redefine what things are local; related paths can be grouped as a collection of paths. Now any rules that propagate assertions about collections of paths are in fact local. In the multiplexer, for example, we might group together the paths DO-1, DO-2, and DO-3. If the device is described in a structural hierarchy, we might use that hierarchy as the basis of the collections. In most cases this will be appropriate.[16] Since we want local propagations to make use of global information, we have considered the idea of "global comments." Global comments are annotations accessible locally, shared by, and referring to, several paths. For example, path DO-1 may have a global comment stating "One of the paths (DO-1 DO-2 DO-3) has the value d2," where each of DO-1, DO-2, and DO-3 have been assigned the value (X, d2). Now GS-5 should only fire when just such a global comment is present on the gate output.

It may be that the problem with this approach lies in the enumeration of rules like GS-5 that propagate sets of values. Clearly, we don't want to write different rules for every possible combination. What is really required is to be able to deduce the appropriate assignments from the existing rules.

Even if these additions are made, however, the problem described in the next section arises.

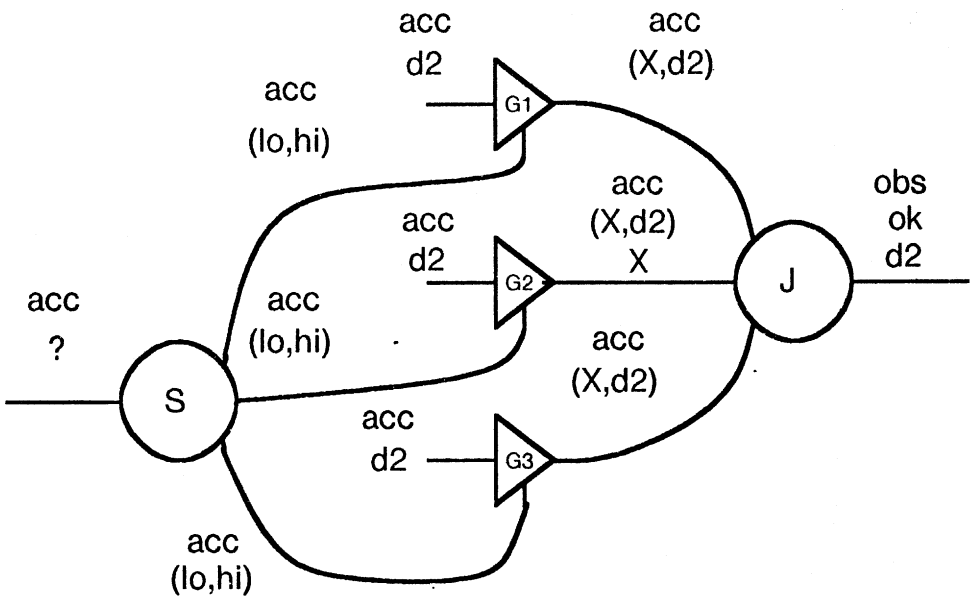
9.1.3 "Accomplish" and "Observe" are not a Complete Set of Goals

Imagine that we have reached a stage of the inquiry design for [DO ? di] at which E-1, E-2, and E-3 all had the value (hi,lo), as shown in Figure 53. No further propagations occur.

We cannot leave the address input unassigned, because the conditions of the inquiry may change depending on its value. For example, if we select d1, then E-1 will be hi, and DO-1 will be d3. E-2 and E-3 will be lo, and DO-2 and DO-3 will be X. Thus, although the value of the address has been left arbitrary, the value actually chosen determined which paths

16. In fact it is hard to imagine cases in which the designers' hierarchy is not appropriate; if such cases were common it would call into question the wisdom of using higher level models in the first place. Nevertheless, it would be nice if the test designer discovered the appropriate global view rather than having the global information built into the structure description.

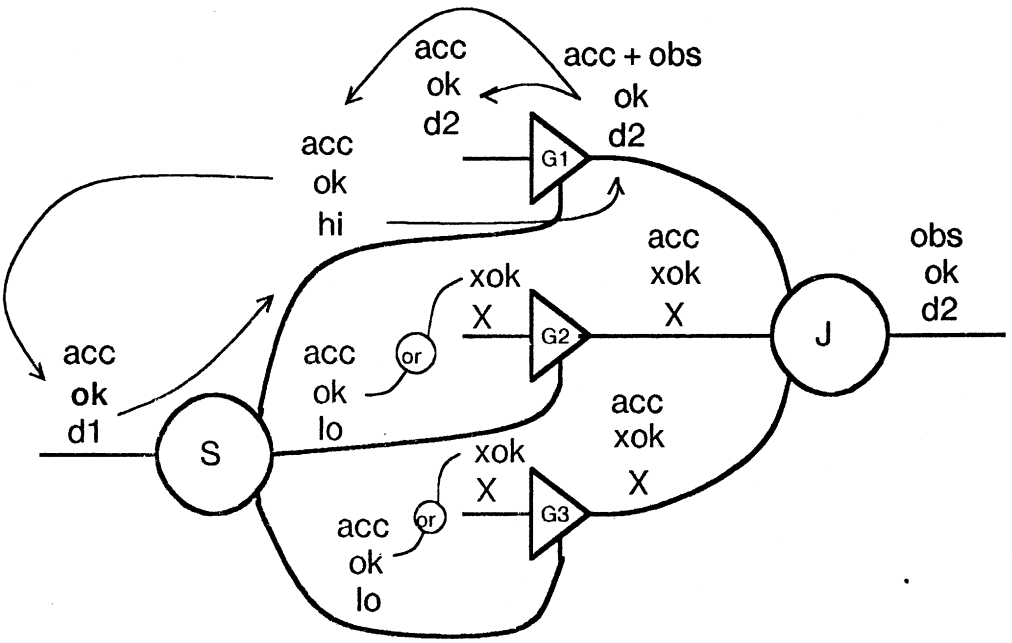
Fig. 53. Result of Applying GS-5



needed to be OK. The conditions that arise in this case are shown in Figure 54.

Unfortunately, conditions have propagated so that the inquiry includes the condition OK on A, suggesting that the whole effort to remove A from the conditions for [DO ? di] was in vain.

Fig. 54. Assignment of Conditions After Choosing (A = d1)



The problem is that we did not "accomplish" d*1* on A, nor "accomplish" the values derived as a consequence of that choice. While E-1, E-2, and E-3, whose values were consequences of that choice, may have conditions due to the assignments on DO-1, DO-2, and DO-3, the path A should not have any conditions.

We need a different goal notation to indicate that A was not forced, but was an unconstrained choice. We also need to modify Condition Rule SC-2 so that the selector input condition is only used when the value is forced. By overriding the existing "accomplish" notation on the selector input with "choice" when choosing a value, we get the desired effect.

9.2 Inquiry Design With State Devices

It is important for a diagnostic generation system to handle devices with state. The system's inquiry improvement and test ordering phases seem to require no adaptations to do this. Currently the inquiry design phase of the system can design some of the inquiries for devices containing the memory primitive, using the notion of snapshots to accomplish propagation through time, as described in section 5.6. But the search paradigm used currently fails when there is more than one state terminal with a value to be accomplished.

Figure 55 shows a two-way addressable memory. Consider designing an inquiry that tests whether the path E1 can transmit 1o. We finish propagating at time *t* to find that we must accomplish d*n-1* at state terminal S1 at time *t-1*, and X at state terminal S2 at time *t-1*. For simplicity of presentation the conditions are not shown.

Propagating backward one step in time, it appears that there is no way to accomplish both goals at *t-1*, since the behavior of the device is such that only one memory device can be written into per time step. We know, however, that both goals need not be accomplished simultaneously, since the memory devices preserve the values at their state terminals.

The system could be made to rely on the persistence of state values, never trying to accomplish more than one such goal during any time step. This is similar in spirit to the restriction that only one output may ever be observed.

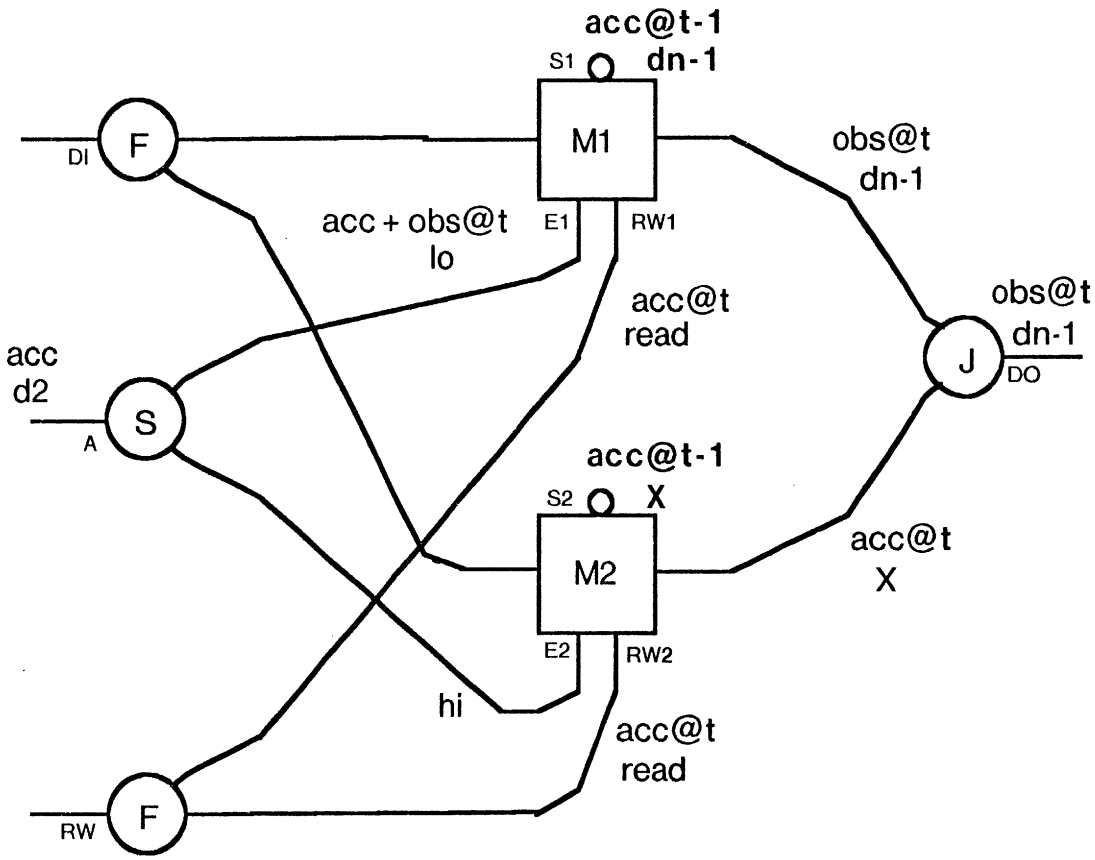
9.3 Diagnostic Design as a Problem

More important than these shortcomings in the propagation and search machinery, the system must also be broadened.

First, it is clear that the vocabulary of devices is extremely limited. It must be extended to include computational devices such as adders and shifters. This is not to propose that every possible device will have its own primitive. Rather, the issue is to identify those primitives that classify hardware structures on the basis of their testing requirements.

Second, while faults on paths is a good place to start on the problem, the possibility of faults in devices must also be considered. As it stands, our devices are simple enough that to include them as possible sources of faults would make the condition lists longer without changing the ordering information significantly. We anticipate that just as we found it

Fig. 55. Assignments in an Addressable Memory at Time t



necessary to have a hierarchy of conditions on paths, we will also find it useful to have a hierarchy of conditions on devices. For example, a multiplier represented as a primitive might be used to multiply by 1, in effect passing data on to another part under test, before it had been fully tested. The appropriate condition for the multiplier would be that it "passes data" correctly.

Third, we must finish the task of generating a runnable diagnostic. Up to now we have used structural information only to modify path conditions; we will also need to use structural information to map our models of paths and devices onto the underlying implementation so as to generate test routines for the real hardware.

10. Conclusion

Recall now the original 4x4 memory example. We were able to figure out the sequence of tests that constituted a diagnostic for that piece of hardware simply by knowing that the address lines selected one of the registers and routed its data to or from it. Among other things, this told us that we should not test the registers until verifying the addressing lines, and that we could test the input and output buses independently of any single registers by iterating over several values of the address.

The system we have described uses the information path model, the notion of an inquiry, and the principle of minimizing conditions to perform the same task. The information path model, although still crude at this point in its vocabulary, is a structural and functional representation of the multiplexer that seems to be appropriate for designing diagnostics. The system views the multiplexer as a structure composed of primitive functions, and plans the diagnostic in a general way. By abstracting away from the "bits and gates" level of description used by conventional test generation approaches, we are able to discover structural regularities and dependencies, so as to plan a diagnostic with the goal of achieving resolution. The principle of minimizing conditions, and the ways that this can be done, work equally well at any level of detail.

Having decided that the individual inquiries that make up the diagnostic have sets of conditions, the problem is then to generate these sets. This problem is approached by augmenting the traditional path sensitization technology. Knowledge of the structure and behavior of the device under test is brought to bear; device-specific rules express the relationship of goals to appropriate values and the conditions that result. Propagation of constraints is shown as a natural way of expressing and using the rules. The inquiry design phase uses information about the behavior of the devices, a simple fault model, and the notion of sensitization to create sets of candidate inquiries that are analyzed and combined to make tests with explicit dependencies.

The conditions on the individual inquiries are then reduced using the single point of failure assumption, and the resulting tests are ordered so as to reduce their conditions. Once the tests have thus been ordered and their internal structure as sequences of inquiries have been defined, the diagnostic can be derived from this representation by translating the inquiries using implementation information.

References

[Batali 81]
Batali, J, *An Introduction to DPL*, AIM-598, MIT 1981.

[Breuer 76]
Breuer M, Friedman A, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.

[Breuer 79]
Breuer, Melvin A. "New Concepts in Automated Testing of Digital Circuits," in "Computer Aided Design of Digital Electronic Circuits and Systems," North-Holland Publishing Company, Brussels and Luxembourg 1979, pp 57-80.

[Davis 82]
Davis R, Shrobe H, Hamscher W, Wieckert K, Shirley M, Polit S, *Diagnosis Based on Description of Structure and Function*, Proceedings AAAI-82, pp. 137-142.

[Franz 81]
Foderaro J, Sklower K, *The FRANZ LISP Manual*, UC Berkeley, September 1981.

[Lai 81]
Lai, Kwok-Woon, *Functional Testing of Digital Systems*, PhD Thesis, Carnegie-Mellon University, Department of Computer Science, December 1981. Technical Report CMU-CS-81-148.

[Roth 67]
Roth, JP, WG Bouricius, and PR Schneider, *Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits*, IEEE Transactions on Electronic Computers, Volume EC-16, pp 567-580.

[Roth 80]
Roth, John Paul, *Computer Logic, Testing, and Verification*, Computer Science Press, 1980, Chapter 3.

[Rutman 72]
Rutman, Roger A, "Fault-Detection Test Generation For Sequential Logic by Heuristic Tree Search," IEEE Computer Research Paper # R-72-187.

[Steele 80]
Steele, Guy L, "The Definition and Implementation of a Computer Programming Language Based on Constraints," Artificial Intelligence Laboratory, AIM-595, Cambridge: MIT, August 1980.

Appendix I - Notation

We use two types of expressions: expressions concerning values and expressions concerning conditions.

(<path> = <value>) is a value clause. E.g. (DO-1 = di) means that the path DO-1 is transmitting the value d*i*.

(<condition> <path>) is a condition clause. E.g. (ok A) means that path A is OK. Another example of a condition clause is (or (ok CTL) (xok IN)), meaning that either path CTL is OK or path IN is XOK.

The following notation is used to describe inquiries.

[<path> ? <value>] represents the focus of an inquiry. [IN-1 ? d2] is an inquiry to see whether path IN-1 transmits the value d2.

(<path> ? <value>) describes the test to be done on an observable output. (OUT ? d2) means that the inquiry should implicate if anything other than d2 is observed on path OUT.

CS-TR Scanning Project
Document Control Form

Date : 2/15/96

Report # AI-TR-707

Each of the following should be identified by a checkmark:
Originating Department:

- [x] Artificial Intellegence Laboratory (AI)
- [ ] Laboratory for Computer Science (LCS)

Document Type:

- [x] Technical Report (TR)
- [ ] Technical Memo (TM)
- [ ] Other: _____

Document Information

Number of pages: 66 (74-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

- [x] Single-sided or
- [ ] Double-sided

Intended to be printed as :

- [ ] Single-sided or
- [x] Double-sided

Print type:

- [ ] Typewriter
- [ ] Offset Press
- [ ] Laser Print
- [ ] InkJet Printer
- [ ] Unknown
- [x] Other: _____

Check each if included with document:

- [x] DOD Form (2)
- [x] Funding Agent Form
- [x] Cover Page
- [ ] Spine
- [ ] Printers Notes
- [ ] Photo negatives
- [ ] Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP:	(1-66) UN#'ED TITLE, ABS, ACK, CONTENTS (2), FIGURES (2), INTRO, 2-59
	(67-74) SCANCONTROL, COVER, FUNDING AGENT, DOD (2), TRGT'S (3)

Scanning Agent Signoff:

Date Received: 2/15/96 Date Scanned: 2/20/96 Date Returned: 2/22/96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 707	2. GOVT ACCESSION NO.	3. RECIP ER AD-A131859
4. TITLE *(and Subtitle)* Using Structural and Functional Information in Diagnostic Design		5. TYPE OF REPORT & PERIOD COVERED technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Walter Hamscher		8. CONTRACT OR GRANT NUMBER*(s)* N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE June 1983 13. NUMBER OF PAGES 68
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Diagnostics
Diagnosis
Hardware Troubleshooting
Test Generation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

We wish to design a diagnostic for a device from knowledge of its structure and function. The diagnostic should achieve both coverage of the faults that can occur in the device, and should strive to achieve specificity in its diagnosis when it detects a fault. A system is described that uses a simple model of hardware structure and function, representing the device in terms of its internal primitive functions and connections. The system designs a diagnostic in three steps. First, an extension of path sensitization is used

(Over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

to design a test for each of the connections in the device. Next, the resulting tests are improved by increasing their specificity. Finally, the tests are ordered so that each relies on the fewest possible connections. We describe an implementation of this system and show examples of the results for some simple devices.

REPORT DOCUMENTATION PAGE

1 REPORT NUMBER
AI-TR 707

4 TITLE (and Subtitle)
Using Structural and Functional Information in Diagnostic Design

5 TYPE OF REPORT & PERIOD COVERED
Technical Report

7 AUTHOR(s)
Walter Hamscher

8 CONTRACT OR GRANT NUMBER(s)
N00014-80-C-0505

9 PERFORMING ORGANIZATION NAME AND ADDRESS
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

11 CONTROLLING OFFICE NAME AND ADDRESS
Advanced Research Projects Agency
1400 Wilson Blvd
Arlington, Virginia 22209

12 REPORT DATE
June 1983

13 NUMBER OF PAGES
66

14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Office of Naval Research
Information Systems
Arlington, Virginia 22217

15 SECURITY CLASS. (of this report)
UNCLASSIFIED

15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)
Distribution of this document is unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18 SUPPLEMENTARY NOTES
None

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)
Diagnostics
Diagnosis
Hardware Troubleshooting
Test Generation

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

We wish to design a diagnostic for a device from knowledge of its structure and function. The diagnostic should achieve both coverage of the faults that can occur in the device, and should strive to achieve specificity in its diagnosis when it detects a fault. A system is described that uses a simple model of hardware structure and function, representing the device in terms of its internal primitive functions and connections. The system designs a diagnostic in three steps. First, an extension of path sensitization is used

(over)

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 2/20/1996

M.I.T. Libraries
Document Services