Report 81-20
Stanford -- KSL

Scientific DataLink

The Use of Design Descriptions in Automated Diagnosis.
Michael R. Genesereth,
Dec 1981

card 1 of 1

Stanford Heuristic Programming Project
Memo HPP-81-20

First Version December 1981
Current Version January 1984

The Use of Design Descriptions in Automated Diagnosis

by

Michael R. Genesereth

Department of Computer Science
School of Humanities and Sciences
Stanford University

Abstract: This paper describes a device-independent diagnostic program called DART. DART differs from previous approaches to diagnosis taken in the Artificial Intelligence community in that it works directly from design descriptions rather than MYCIN-like symptom-fault rules. DART differs from previous approaches to diagnosis taken in the design automation community in that is more general and in many cases more efficient. DART uses a device-independent language for describing devices and a device-independent inference procedure for diagnosis. The resulting generality allows it to be applied to a wide class of devices ranging from digital logic to nuclear reactors. Although this generality engenders some computational overhead on small problems, it facilitates the use of multiple design descriptions and thereby makes possible combinatoric savings that more than offsets this overhead on problems of realistic size.

1. Introduction

Continuing advances in the technology of design and manufacturing have led to artifacts of unprecedented complexity. Devices like VLSI chips, nuclear power plants, and jet aircraft are among the most complex physical objects ever created by man. Unfortunately, the physical components from which these devices are built are subject to failure; and, given current design practices, the failure of a single component can lead to the malfunction of an entire device. The key disadvantage of complexity is that it makes the diagnosis of such component failures more difficult.

The DART program is an automated diagnostician aimed at dealing with this complexity. The program is intended for use in conjunction with a tester that can manipulate and observe a malfunctioning device, as suggested by figure 1. The diagnostician accepts from the tester a description of an observed malfunction, prescribes tests and accepts the results, and ultimately identifies the faulty components responsible for the malfunction.

DART was developed in the context of moderately successful work on medical diagnosis, as exemplified by such programs as INTERNIST [Pople 1975] [Pople 1977], and MYCIN [Shortliffe]. However, there's a difference. These medical diagnosis programs both use "shallow" theories of human pathophysiology in the form of "rules" that associate symptoms with possible diseases. The DART program contains no rules of this form. Instead, it works directly from a "deep" theory consisting of information about intended structure (a device's parts and their interconnections) and expected behavior (equations, rules, or procedures that relate the device's "inputs", "outputs", and "state").

An important advantage of this approach is that it greatly simplifies the task of building diagnosticians for new devices. If a designer uses a modern computer-aided design system like PALLADIO [H. Brown, Tong, Foyster], then when he is done there is an on-line description of his design. This description can be passed as data to the DART program to diagnose its faults. Similarly, this design information can be passed to a program to simulate the device, a program to verify and evaluate the design, a program to generate testing codes, and a program to generate fabrication instructions. See figure 2.

The idea of using design information in automated diagnosis is hardly a new one. Over the years a number of test generation algorithms have been proposed im the domain of computer hardware [Breuer, Friedman], the most well-known of which is the d-algorithm [Roth et al]. The primary disadvantage of these algorithms is their specificity. For example, the d-algorithm is based on boolean algebra (or, more precisely, Roth's "d-calculus"), and so it is applicable only to devices whose behavior can be characterized in terms of ones and zeros.

By contrast DART uses a device-independent language for design description and a device-independent diagnostic procedure. All device-dependent information is contained in the design descriptions it uses. Because of this generality, DART can

diagnose a wider class of devices than the d-algorithm, including non-digital and non-electronic devices.

Interestingly, this generality can also promote enhanced efficiency. The device-independence of the design language and diagnostic procedure makes it easy to apply the program at multiple levels of abstraction and thereby exploit the hierarchy inherent in most computer system designs. The power of the design language and diagnostic procedure makes it easy to use symbolic constraint propagation techniques [Sussman, Steele] [Stefik] and thereby avoid unnecessary combinatorics.

This paper describes the DART program in detail and discusses the role of design descriptions in diagnosis. Section 2 shows how design can be formally described, and section 3 formalizes the description of actual devices. The concepts of fault, symptom, and diagnosis are defined in section 4, and the DART program is described in section 5. Section 6 discusses the interrelationship between the choice of design description and the completeness and efficiency of diagnosis. The conclusion discusses the state of implementation and testing of DART, offers some directions for future work, and summarizes the key points of the paper. This paper is an expanded version of a previous publication [Genesereth].

2. Design Description

In this paper a *design* is broadly defined as any arrangement of the world that achieves a desired result for known reasons. The arrangement can be artificial (e.g. a computer system), or it can be natural (e.g. the human circulatory system). The desired result can be a specific situation, or it can be a behavior that varies from one situation to another. What characterizes a design is that its result is explanable in terms of an assumed theory of the world.

A digital circuit like the "full adder" is a good example. A full adder is essentially a one bit adder with carry in and carry out, and it is usually used as one of n elements in an n bit adder. A graphical representation of its design is given in figure 3. It has three inputs and two outputs and consists of two "xor" gates (X1 and X2), two "and" gates (A1 and A2), and an "or" gate (O1). A tabular representation of its behavior is given in figure 4. In normal operation, the first output (the "sum line") is "on" if and only if an odd number of inputs is "on"; and second output (the "carry" line) is "on" if and only if at least two inputs are "on". Behavioral tables for the components in figure 3 are given in figure 5. Using this information, it is possible to verify that the design in figure 3 achieves the behavior in figure 4.

In DART all descriptions are written as propositions in a variant of prefix predicate calculus augmented with a small set of modal operators. Upper case letters are used exclusively for constants, function symbols, and relation symbols, while lower case letters are used for variables. Prefix universal quantifiers are dropped, and all free variables are universally quantified.

For example, the propositions in figure 6 constitute a design description for the device in figure 3. Each part is designated by an atomic name (e.g. A1). The structural type of each part is declared using type relations, as in the first five

propositions: X1 and X2 are "xor" gates, A1 and A2 are "and" gates, and O1 is an "or" gate. The "inputs" and "outputs" (or "ports") of each device are named using the functions `IN` and `OUT`. For example, `(IN 2 F1)` designates the second input of F1, and `(OUT 2 F1)` designates the second output of F1. Connections are made between the ports of devices. The remaining propositions specify the wiring diagram for F1. The sixth proposition states that the first input to F1 is connected to the first input of X1; the last proposition states that the output of O1 is connected to the second output of F1.

The propositions in figure 7 describe the behavior of a full adder. The function symbol `VAL` is used to designate the signal value on a port at a given time. For example, the proposition `(VAL (IN 2 F1) 3 ON)` states that at time 3, the second input of F1 is "on". The proposition `(VAL (OUT 2 F1) t OFF)` states that the second output of F1 is always "off". The eight propositions in figure 7 capture the eight cases illustrated in figure 4.

The behavior of components can be expressed similarly. The propositions in figure 8 illustrate. The first three propositios describe the behavior of an "and" gate. The next three describe the behavior of an "or" gate. The four thereafter describe an "xor" gate. The final proposition describes the behavior of an ideal connection. A variation of this rule is discussed in section 6.2.

3. Device Description

A *device* is any physical realization of a design. Unfortunately, due to fabrication errors and physical failures, the actual structure of a device may differ from its intended structure, and this can give rise to differences in behavior. The information in a device description comes from a variety of sources.

Theoretical information includes the definitions and theorems used in proving the correctness of a device. For example, in the case of the full adder it would be relevant to include the propositions from figure 8 in F1's device description as well as its design description.

Achievable data are propositions that can be made true by a tester. This can include structural modifications of which the tester is capable, such as part swaps and the addition of testing equipment. More typically, it involves modifications to the environment of a device, such as setting the values of its inputs. For example, the propositions in figure 9 describe a sequence of inputs that a tester has applied to the full adder F1.

Observable data are propositions that can be directly observed by a tester. In some cases the actual structure of a device can be observed. Boards can be visually inspected, and chips can be examined microscopically. However, observations of behavior are more common. For example, the propositions in figure 10 describe the output values resulting from the inputs in figure 9.

Unfortunately, it is not possible to diagnose all malfunctions using just theoretical, achievable, and observable data. And even when it is possible, it is often

impractical. For these reasons it is customary to augment this hard data with some assumptions about the malfunctioning device, as in figure 11.

A common example is the assumption that certain parts of the device's design description are correct. For example, the connections in a circuit may be guaranteed correct and the functionality of the components may be in doubt, or vice versa. In the examples below, all connections are assumed to be correct.

The single fault assumption (SFA) states that there is at most one faulty component in a circuit. This idea can be formalized by stating that the failure of any component implies the functionality of the others. The next five propositions in figure 11 specify the single fault assumption for the components of F1. The single fault assumption can be stated more succinctly as a single proposition that applies to all components, but that rendition is slightly more complicated than the one given here.

The nonintermittency assumption (NIA) states that all devices behave consistently over time. This is patently false in general, for it implies that no part can ever fail. However, it is often a reasonable assumption to make for the duration of a diagnosis. A formal statement of the nonintermittency assumption follows the single fault propositions in figure 11. It says that, if a device with specific inputs has a given output at one time, then given the same inputs it will have the same output at any other time.

Assumptions like these are extremely *important to* the efficiency of a diagnostic procedure like DART as well as to its competence in diagnosing difficult malfunctions. However, it is important to emphasize that they are not *part of* the procedure. They can be included or not at the discretion of the user. This flexibility is essential for situations in which the assumptions are wrong. For example, there may be more than one fault in a device, or the device may be malfunctioning intermittently. The inappropriateness of an assumption usually manifests itself as a contradiction. Davis [Davis] discusses some approaches to relaxing assumptions to deal with such contradictions.

4. Diagnosis

A *fault* is any discrepancy between the actual structure of a device and its design. In DART terms this corresponds to any proposition from the device's design description that is not true of the device itself and, therefore, is inconsistent with its device description. For example, if the component X1 of the full adder F1 were broken, the proposition describing its type would be incorrect.

When a device is faulty, its behavior can differ from what is predicted from its design. In DART terms, a *symptom* is any observable proposition that, when added to a device description, makes it inconsistent with the device's intended behavior. For example, given the achievable data in figure 9, the first observation in figure 10 is a symptom of some fault in F1, because it disagrees with the behavioral description in figure 7.

The goal of *diagnosis* is to determine the fault or faults responsible for a set of symptoms. The diagnosis of a fault can be considered complete if the device description contains sufficient information to prove it. For example, from the information in figures 8, 9, 10, and 11, it is possible to prove `(NOT (XORG X1))`, i.e. that X1 is broken. Of course, a circuit may have more than one fault.

The purpose of diagnostic testing is to obtain sufficient information to complete a diagnosis. Of course, some faults are *undiagnosable* in that there is no set of achievable and observable data that allows them to be distinguished from each other.

The importance of design information in diagnosis can best be seen by thinking of the process as one of theory formation. Starting with a set of data about a device, the goal of diagnosis is to produce a description of its actual structure. Without the design description, diagnosing a device is indistinguishable from designing a device that exhibits the observed behavior. A design description constrains the kinds of theories that the diagnostician can form by forcing it to consider only propositions from the design description or their negations.

5. The DART Program

DART is a device-independent diagnostician that works directly from information about the design of a malfunctioning device. Given a set of symptoms, it generates tests, accepts the results, and ultimately pinpoints at least one fault in the device or an undiagnosable cluster containing a fault.

5.1 Overview

DART begins with a design description for a malfunctioning device and a partial device description. For device F1, the design description would include the propositions in figure 6. A typical device description might include the theoretical propositions and assumptions in figures 8 and 11 together with the achievable and observable propositions from figure 9 and 10 relevant to time 1.

A flowchart for DART appears in figure 12. The program first computes a set of suspect propositions. If this set contains only one element, the diagnosis is done. Otherwise, DART tries to generate a test to discriminate the suspects. If it is successful, the resulting test is executed, its consequences are computed, and the process repeats. Otherwise, the set of suspects is returned as value.

Note that DART does *not* generate a complete diagnostic tree before executing tests. The reason for this is efficiency. Although complete diagnosic planning has the advantage of minimizing the cost of test execution, the computational cost of constructing a complete diagnostic tree is usually prohibitive.

5.2 Computing Suspects

The goal of suspect computation is to identify a subset of the design description within which some fault is guaranteed to lie. In computing a suspect set, DART starts with the observed symptoms and tries to deduce a proposition of the following form,

where each `pi` is a statement from the circuit's design description not already known to be true. Obviously, the smaller the set, the better.

```
(OR (NOT p1) . . . (NOT pn))
```

As an example, consider the symptom in figure 10 relevant to time 1. Since the first output of F1 is not "on" and the output of X2 is connected to it, then the connection rule requires that the output of X2 must not be "on" either. Since the output of X2 is not "on", then either it is not a functioning "xor" gate or it is not the case that its two inputs are "on" and "off". However, the second input must be "off", since it is connected to the third input of F1, and that is known to be "off". If the first input is not "on", then the output of X1 must not be "on", and this could only be the case if X1 is not a functioning "xor" gate, since its inputs are known to be "on" and "off". Thus, either X1 or X2 is broken.

```
(OR (NOT (XORG X1)) (NOT (XORG X2)))
```

One thing to note is that suspect computation is not simply a matter of tracing a design diagram backwards from a faulty output. On the one hand, this can lead to too many suspects. For example, tracing backwards from the second output of F1 leads to four components, yet an error in the second output with the input data above implies there is a fault in one of the three components A1, O1 and A2. X1 may have a fault, but it is not involved in this error and shouldn't be included in the suspect set.

On the other hand, simply tracing backwards can also lead to too few suspects. For example, it would rule out the possibility that components elsewhere in the circuit could have any bearing on the symptom and so would make it impossible to diagnose short circuits.

A second thing to note is that in some situations it is possible to derive several different suspect sets. There may be several symptoms, each giving rise to its own suspect set. For example, given inputs "on", "off", and "on", an error in the first output of F1 would implicate X1 and X2, and a simultaneous error in the second output would implicate X1, A2, and O1. It is also possible to generate several suspect lists from a single symptom. For example, if all three inputs to F1 were "off", an error in the second output of F1 would implicate X1, A2, O1, and A1 through one line of argument, and it would implicate just A2, O1, and A1 through a different line of argument.

When a device description includes the single fault assumption, it is possible to intersect multiple suspect sets. In the first case, this would immediately identify X1 as the culprit. In the second case, it would narrow the suspect set to A1, A2, and O1. Note that in order to get as small a bound as possible on the suspects, it may be necessary to deduce and intersect all suspect sets. Due to the decidability of the suspect computation for most design descriptions, this is usually possible and frequently practical.

5.3 Generating Tests

The goal of diagnostic testing is to gather data to help confirm or disconfirm the propositions in the suspect set. Of course, some data is more valuable than other data in this regard. Test generation is the process of deciding which data to gather.

In DART a *test* consists of zero or more propositions to be achieved and at least one proposition to be observed. One way to generate a test is to assume some set of possible faults, select a set of achievable propositions (e.g. input values), and simulate the circuit to predict an observable (i.e. an output value). The problem with this approach is that the resulting test may not depend on any of the suspects. The diagnostician may be forced to try numerous sets of achievables before coming upon one that supplies any information about the malfunction. This may be okay for small devices like the full adder, but it is unacceptable for large devices like VLSI chips and entire computer systems, where the number of input combinations is astronomically large.

The alternative used by DART is to start with one of the suspect propositions and try to deduce a proposition of the following form, where each of the `ai` is an achievable proposition, where each of the `pi` is a suspect proposition, and where `ob` is an observable proposition.

```
(IF (AND a1 . . . am ob)
    (OR (NOT p1) . . . (NOT pn)))
```

As an example, consider the problem of computing a test to discriminate the suspects computed in the last section. DART begins the test generation process with the tautologies `(IF (XORG X1) (XORG X1))` and `(IF (XORG X2) (XORG X2))`, one for each suspect. Applying resolution residue to the former of these leads to the test shown below. The right hand side of the tautology unifies with the behavioral rules for X1, in particular the rule stating that, if the gates's two inputs are "on" and "off" respectively, its output is "on". This value propagates to the input of A2; and, if the third input to F1 is "on", the value also propagates to the output of A2. It then propagates to the input of O1, to the output of O1, and finally to the second output of F1. In summary, if the inputs to F1 are "on", "off", and "on", the second output must be "on" so long as X1 is working properly. This is equivalent to saying that, if the inputs are "on", "off", and "on" and the second output is *not* "on", X1 is broken.

```
(IF (AND (VAL (IN 1 F1) t ON) (VAL (IN 2 F1) t OFF) (VAL (IN 3 F1) t ON)
         (NOT (VAL (OUT 2 F1) t ON)))
    (NOT (XORG X1)))
```

Of course, to have diagnostic value at all the outcome of a test must not be predictable, i. e. it must provide new information. Once DART generates a test, it is checked for novelty by trying to prove the propositions shown below. If it fails to prove either proposition, then the test provides new information.

```
(IF (AND a1 . . . am) ob)
(IF (AND a1 . . . am) (NOT ob))
```

The proposition below is an example of a test that DART might generate in this situation that provides no new information. The reason is that the output of the test is completely predictable. By the single fault assumption, A1 and O1 are operational.

Since the two inputs to A1 are "on", its output and the second input to O1 must be "on". But then the output of O1 and, therefore, the second output of F1 must be "on", whether or not the first input is "on". In short, the test is not "sensitive" to any possible failure and so provides no new information.

```
(IF (AND (VAL (IN 1 F1) t ON) (VAL (IN 2 F1) t ON) (VAL (IN 3 F1) t ON))
         (NOT (VAL (OUT 2 F1) t ON)))
    (NOT (XORG X1)))
```

5.4 Drawing Conclusions

Once DART generates an acceptable test, it instructs the tester what to achieve and what to observe. When the test's results are obtained, it then stores them in its device description. For example, if the test derived in the last section were applied to F1 and the corresponding result were observed, DART would store the propositions from figures 9 and 10 relevant to time 2.

DART also does a limited amount of deduction from its test results in an attempt to prune the set of suspects. The data in figures 9 and 10 make a simple example. The achievable and observable propositions added to the device description together with the test proposition from the last section allow DART to conclude that the "xor" gate X1 is faulty. Since this prunes the suspect set to a single element, the diagnosis is complete. In general, this would not be the case, and DART would continue to generate and execute tests until it failed to generate a test or until it eliminated all but one suspect.

5.5 Resolution Residue

The key to DART's generality is its use of a device-independent inference procedure in all phases of the diagnostic process. The computation of suspects, generation of tests, and updating of suspects are all done by proving appropriate propositions. The inference procedure is a variation of resolution [Robinson] called *resolution residue* [Finger, Genesereth].

Resolution residue is similar to resolution in that it operates on propositions in conjunctive normal form and uses the resolution rule of inference. However, unlike resolution, it is a direct proof procedure rather than a refutation method. The procedure begins with propositions known to be true and terminates only when it deduces a proposition that satisfies two criteria.

The first criterion concerns the form of the proposition. Resolution residue succeeds only when it deduces a disjunction in which each literal is in one of a small number of prespecified classes. In computing suspects, each literal must be the negation of a proposition from the malfunctioning device's design description, as described in section 5.2. In generating tests, each literal must be the negation of an achievable proposition, an observable proposition or its negation, or the negation of a proposition from the design description, as described in section 5.3. In drawing conclusions from the results of a test, each literal must be the negation of a proposition from the design description, as with suspect computation.

The second termination criterion is consistency. Each literal in a deduced proposition must be consistent with the global data base and the other literals in the proposition. This is important in computing suspects and drawing conclusions to be sure the result is not simply tautologous. It is important in generating tests to be sure that the conditions of the test can be achieved. Evaluating the consistency of a literal is a nonmonotonic deduction accomplished by trying to prove its negation. If the proof attempt terminates unsuccessfully, the literal is consistent. Of course, this consistency test may not terminate, in which case resolution residue fails. Fortunately, the problems that arise in diagnosing most computer hardware faults are decidable.

Figures 13-15 present examples of resolution residue in diagnosis. Each line is the result of resolving the previous line with the proposition named on the right. In some cases steps are skipped, but all propositions used in the omitted resolutions are named on the right. Figure 13 shows the derivation of the suspect set described in section 5.2 and illustrates the use of the single fault assumption in exonerating all non-suspects. Figure 14 shows the derivation of the test described in section 5.3. Figure 15 shows the derivation of the conclusion described in section 5.4. In all cases the derived propositions are consistent with each other and the propositions in figures 8-11.

An important concern in using resolution in diagnostic reasoning is its potential for proliferating useless deductions. In order to minimize this problem, DART employs a variety of control schemes to focus its effort. It recognizes and separately solves independent subproblems. It uses the "unit preference" strategy to draw simple conclusions before introducing disjunctions. It uses branchiness as a criterion in ordering subproblems [Smith, Genesereth 1983a]. It prunes deduction paths on the basis of solution set sizes [Smith, Genesereth 1983b]. It interleaves consistency checking with residue computation. Finally, it caches partial residues for use in subsequent computations [Lenat].

6. Design Description and Diagnosis

The primary problem in diagnosing devices directly from design information is computational cost. In the world of digital circuits, the cost of executing a test is usually small and the number of tests needed to pinpoint a fault is at worst linear in the number of components [Goel]. However, the cost of generating appropriate tests usually grows nonlinearly, and in general the problem is NP-complete [Ibarra and Sahni]

On the other hand, diagnosis is relative to the design description for the malfunctioning device, and by using a different design description it is frequently possible to achieve dramatic improvements in the efficiency of test generation. Fortunately, design descriptions of this sort are frequently available. Most designers begin with high-level design descriptions and successively refine them until the details are complete. The design descriptions produced in this process frequently contain sufficient information to generate diagnostic tests but suppress enough detail that the cost of test generation is far less than it would be if the full-blown descriptions were used instead.

This section describes some properties of design descriptions that lead to substantial efficiency improvements. *The advantage of the DART program over procedures like the d-algorithm is that it works with a wide enough class of design descriptions that it can exploit these properties to achieve enhanced efficiency.*

6.1 Structural Abstraction

A structural abstraction of a design description is one in which much of the structural detail has been suppressed. The most common example is structural hierarchy. The structure of a complex device is often described in terms of high-level components, whose internal structure either is not specified at all or is specified separately. For example, the structure of an arithmetic circuit like the one in figure 16 is easily described in terms of adders and multipliers. The adders and multipliers can be separately described in terms of their subcomponents, and so on until one reaches the level of gates, transistors, etc.

The advantage of structural abstraction for diagnosis is that it is often possible to diagnose faults without considering the suppressed structural detail. For example, in digital electronics it is usually adequate to diagnose a device to the level of chips or boards rather than the level of individual gates, and this can often be done without knowing their substructure.

Even when the diagnosis must be done to the level of gates, the structural hierarchy can be valuable. For example, it is possible to diagnose the device in figure 16 at a high level of abstraction to determine the major subcomponent in which the fault lies (e.g. the adder AA). This subcomponent can then be diagnosed to identify the fault at the next lower level (e.g. the full adder F1), and so on until the lowest level failure is determined (e.g. X1's output line stuck "off"). By doing the diagnosis hierarchically, the number of components under consideration at any one time can be kept small; and, even though the higher level components are often quite complex, the cost of test generation remains manageable.

Assuming that at each level the cost of diagnosis is some function F_i and the number of possible faults is N_i, the overall cost of hierarchical diagnosis can be expressed as follows, where H is the number of levels.

$$\sum_{i=1}^{H} F_i(N_i)$$

The computational advantage of hierarchical diagnosis is most apparent when the number of gates and the cost function are constant from level to level. Then, for a device of H levels, with N possible faults at each level, the cost of non-hierarchical diagnosis is $F(N^H)$, whereas for hierarchical diagnosis, the cost is only $H*F(N)$. This saving is of special importance when the cost function F is non-linear, but even in the case of linear cost the hierarchical approach still offers a logarithmic advantage.

At first glance this efficiency gain may look illusory. Components at higher levels

in the structural hierarchy have more complex behavior than components at lower levels; and, consequently, the diagnostic cost function F_i is not always constant. For example, going from the gate level to the arithmetic level in figure 16 decreases the number of parts but increases their complexity. Each component at the gate level has only four possible inputs, whereas each component at the arithmetic level has sixteen.

Fortunately, the increase in behavioral complexity as one ascends the structural hierarchy seldom grows as fast or faster than the decrease in structural complexity. So long as this is true, there is computational advantage in using the structural hierarchy. In some cases, the saving comes simply from the use of behavior tables for the higher level components (e.g. the bit tables for a full adder), rather than computing their behavior from lower level components. In other cases, the saving comes from the exploitation of important behavioral properties of higher level components as described in the next sections.

One problem with diagnosis using structural abstractions is that the loss of information can lead to undiagnosability. As an example, consider the circuit in figure 17a. Without any information about the structure of the "negation" device, this circuit cannot be diagnosed. However, given the additional information in figure 17b, it's easy to generate discriminatory tests for many additional faults.

6.2 Behavioral Abstraction

A behavioral abstraction of a design description is one in which much of the behavioral detail has been suppressed. Behavioral abstractions for many designs are determined by the structural hierarchy. The behavior of components at one level in the structural hierarchy is often described in different terms from the behavior of components at other levels. For example, the behavior of gates is best described in terms of "ons" and "offs" rather than voltages. The behavior of adders and multipliers is best described in terms of numbers rather than "ons" and "offs". The behavior of computer networks is best described in terms of packets rather than characters.

A common example of behavioral abstraction is the consolidation of sets of values into single values. For example, in going from the level of transistors to gates, it is reasonable to divide the voltage continuum into discrete values. Similarly, the inputs and outputs of arithmetic devices can be characterized as prime and nonprime, even or odd, positive or negative. Considerable savings can be realized by computing with abstract values of this sort rather than by enumerating the lower level values they summarize.

Another case of behavioral abstraction is the disregard of detail for the sake of simplification. A good example is the rule for the behavior of connections given in figure 8. The rule states that if a port x is connected to a port y and the signal value on port x is z, then the signal value on port y is z. As stated, the rule ignores the problem of contradictory values that might arise in the presence of shorts. This makes the tasks of suspect computation and test generation much easier, because it is unnecessary to consider connections other than those explicitly described in the

design.

Unfortunately, abstracting away the dependence of signal values on other connections makes the diagnosis of shorts impossible. In computing suspects with the connection rule in figure 8, a program like DART has no way of hypothesizing the presence of undesirable connections, since their effect isn't documented. Of course, it is possible to describe the behavior of connections correctly and thereby cure this problem. One need only add a nonmonotonic condition to the rule to make it dependent on the absence of connections other than those in the design. The disadvantage is that this drastically increases the number of suspects for any fault, since the rule would be applicable everywhere. Davis [Davis] describes some ways to keep this growth manageable.

6.3 Constraints

Another way of gaining efficiency in test generation is by formulating a device's design description in terms of interesting functions and relations rather than strictly in terms of values. For example, the behavior of the arithmetic devices in figure 16 can be described in terms of mathematical functions like addition and multiplication, as shown below, rather than in terms of value tables, like those in figure 8.

```
(IF (AND (ADDER d) (VAL (IN 1 d) t x) (VAL (IN 2 d) t y) (= (+ x y) z))
    (VAL (OUT 1 d) t z))
```

The advantage of formulations like this is that these functions and relations can be used to encode constraints on a device's behavior, symbolic constraint propagation techniques can then be used in place of exhaustive enumeration of possibilties.

Consider, for example, the circuit in figure 16 and assume that for test generation purposes it is necessary to get a 4 on the first output and a 6 on the second. One way of doing this is to work backwards from the desired values. For example, if the inputs to the adder A1 were 4 and 0, this would produce the 4 desired on the output. The consequences of this choice can then be propagated around the circuit. In this case, in order for there to be a 0 on the second input to A1, there must be a 0 on the output of M2 and a 0 on the first input to A2. In order for the output of A2 to be 6, the second input to A2 and the output of M3 must be 6. This can be accomplished by seeting the inputs to M3 to 6 and 1. In order for there to be a 6 on the first input to M3, there must be a 6 on the second input to M2. In order to get a 0 on the output of M2, there must be a 0 on its first input and a 0 on the second input to M1. However, this means that the output of M1 must be 0, in contradiction with its previous setting to 4. In other words at least one of the arbitrary choices made at A1 and M3 must be undone and other values tried, until a consistent value assignment is found.

An alternative to this sort of exhaustive enumeration is the posting, propagation, and satisfaction of symbolic constraints. For example, in this case, rather than selecting specific values at each choice point, one could designate those values by variables and record appropriate constraints among them, as suggested in figure 18. These symbolic values can then be propagated and new constraints can be posted,

until all relevant lines have been visited. The constraints accumulated during this process can then be solved to produce values for the variables.

In this approach, no backup is necessary in generating the constraints, and the constraints can often be solved quite efficiently. The cost of test generation without constraint propagation is exponential in the number of choices that must be made. For a device with N choice points and B alternatives at each point, the cost is B^N. Using symbolic constraint propagation, the search of this space is replaced by the problem of solving a set of simultaneous equations in N variables. If the equations are linear, this can be done in time proportional to N^3.

Of course, the constraints produced during test generation are not always this inexpensive to solve; and when this is the case the value of constraint propagation is open to debate. On the one hand, constraint propagation is good because it breaks the test generation problem into two parts, viz. the basic search cost and the cost of propagation. Although the basic search cost may not be improved, the cost of propagation is paid only once. On the other hand, the overhead of propagating symbolic constraints may outweigh this saving. The merit of constraint propagation appears to be situation dependent.

6.4 Conditional Values

Another design description reformulation of particular utility in generating tests for circuits is the use of "conditional values" in the behavioral description of a design. A true-false conditional value consists of two signal values and an implicit condition. The presence of a true-false conditional value on a port means that the port has the first value if the condition is true, and it has the second value if the condition is false. An n-way conditional value consists of n signal values and an implicit condition with n possible outcomes, and its presence on a port means that the port has the first value if the first outcome is true, etc.

Conditional values are of particular use in generating bidirectional tests. A bidirectional test is one that is guaranteed to provide information about a set of faults whether it succeeds or fails. If it succeeds, the conclusion of the test can be assumed to be true. If it fails, the conclusion can be assumed false. In other words, the implication is bidirectional.

The d-algorithm provides an interesting implementation of this technique. Instead of propagating just "ons" and "offs", the d-algorithm employs an additional pair of symbols for each suspect. In order to generate a test for a component, one of these symbols is placed on the output line of the component. If the value of the line is supposed to be "on" when the component is functioning properly, the "positive" symbol (e.g. d) is placed on the line. If the output is supposed to be "off", the "negative" symbol (e.g. $\underline{d}$) is used instead. These values are then propagated through the circuit. The occurrence of a positive symbol on a line means that that line should be "on" if the component corresponding to the symbol is working properly, and it should be "off" otherwise. The occurrence of a negative symbol means that the line should be "off" if the component is functioning properly and vice versa. The arrival of a positive symbol at a primary output signals a bidirectional test for the associated

component. The arrival of a positive symbol for one component and a negative symbol for another is discriminatory.

The advantage of conditional values like these is that they allow several unidirectional tests to be generated simultaneously. The cost of generating these tests is not substantially greater than the cost of generating a single test, but together they yield more information. Furthermore, conditional values make it easy for a diagnostician to recognize evolving bidirectional tests during test generation and give them control priority.

The disadvantage of conditional values is that the behavioral description of each component must be augmented to include propagation rules for these new symbols. Fortunately, this can be done automatically. Unfortunately, it expands the number of rules that a diagnostician must store and use. In preliminary experiments the computational benefit of this approach has not warranted the cost of these modified descriptions. However, the technique looks promising and warrants further study.

7. Conclusion

The DART program has been implemented and tested on a variety of examples. The implementation was done in MRS [Genesereth, Greiner, Smith]. The test examples include simple circuits like the one in figure 3, more complex devices like the teleprocessing system of the IBM 4331, and non-electronic devices like the cooling system of a nuclear reactor. In all cases the program was able to generate tests and diagnose underlying faults. The time to diagnose each case was on the order of seconds or minutes. While this is not particularly good for small circuits, it is excellent for complex devices.

One limitation of DART in its present form is that it doesn't take into account the cost of tests. Some tests are more expensive than others. For example, it's easier to check a status light than to measure a voltage. It's easier to get information at a system terminal than to pore through a core dump. Unfortunately, the evaluation of test cost can be quite difficult because of dependencies on sequencing and grouping. For example, once a file has been deleted, it may be expensive or impossible to recreate. Once a piece of test equipment has been attached to a port, it's just as easy to get a set of values as it is to get an individual value.

DART also doesn't into account the diagnostic value of tests, beyond the novelty check described in section 5.3, even though this can be extremely important in reducing the overall cost of diagnosis.

While DART is faster than the d-algorithm in generating tests for complex devices, computational cost remains a problem. Not all design descriptions are tuned for the task of diagnosis, and a bad description can lead to substantial inefficiency. One way of dealing with this problem is to automate the reformulation of design descriptions into more useful forms, like those suggested in section 6.

A more drastic way of dealing with inefficiency is to "compile" design descriptions into MYCIN-like symptom-fault rules. If it is possible to diagnose a

device automatically, it should be possible to create a set of diagnostic tests automatically. The cost of this approach can be formidable since it must take into account all possible faults. However, the savings can also be dramatic, and the approach warrants further investigation.

One final issue worth mentioning is DART's absolute dependence on the existence of design information. Missing or inappropriately represented information can lead to extreme inefficiency and undiagnosability. While adequate information about the design of a device is usually available in the heads of its designers, it is a significant chore to express it in a machine-readable form. The problem can be mitigated by the creation of design tools that facilitate the entry of useful design descriptions. Unfortunately, design information is sometimes unavailable outside of the organization responsible for the design; and, whenever this is the case, the DART program is of little use.

In summary, the key contribution of this research is the DART program. DART differs from previous approaches to diagnosis taken in the Artificial Intelligence community in that it works directly from design descriptions. DART differs from previous approaches taken in the design automation community in that is more general and in many cases more efficient. DART uses a device-independent language for describing devices and a device-independent inference procedure for diagnosis. The resulting generality allows it to be applied to a wide class of devices ranging from digital logic to nuclear reactors. Although this generality engenders some computational overhead on small problems, it makes possible combinatoric savings that more than offsets this overhead on problems of realistic size. For these reasons, the DART algorithm appears to be a promising way of coping with the increasing complexity of modern designs.

Acknowledgements

The idea of applying this research to computer hardware was first suggested by Ed Feigenbaum. Bob Joyce and Narinder Singh experimented with an early implementation of DART and explored many of the techniques for improving efficiency. Randy Davis critiqued the work at an early stage and pointed out the need for physical information in computing suspects. The work was done with the collaboration and support of the IBM Palo Alto Scientific Center and the Fairchild Laboratory for Artificial Intelligence Research. Additional funding was provided by the Office of Naval Research under contract number N00014-81-K-0004.

References

M. A. Breuer, A. D. Friedman: *Diagnosis and Reliable Design of Digital Systems*,Computer Science Press, 1976.

H. Brown, C. Tong, G. Foyster: "PALLADIO: An Exploratory Environment for Circuit Design", IEEE Computer, December 1983.

R. Davis: "Diagnosis Based on Description of Structure and Function", Proceedings of AAAI-82, August 1982, pp 137-142.

J. Finger, M. R. Genesereth: "RESIDUE - A Deductive Approach to Design", HPP-83-46, Stanford University Heuristic Programming Project, December 1983.

M. R. Genesereth, R. Greiner, D. E. Smith: "MRS - A Meta-Level Representation Ssytem", HPP-83-28, Stanford University Heuristic Programming Project, 1983.

M. R. Genesereth: "Diagnosis Using Hierarchical Design Models", Proceedings of AAAI-82, August 1982, pp 278-283.

P. Goel: "Test Generation Cost Analysis and Projections", 17th Design Automation Conference Proceedings, June 1980.

O. H. Ibarra and S. Sahni: "Polynomially Complete Fault Detection Problems", IEEE Trans. on Computers, Vol C-24 No 3, March 1976, pp 242-250.

D. B. Lenat, F. Hayes-Roth, P. Klahr: "Cognitive Economy", HPP-79-15, Stanford University Heuristic Programming Project, June 1979.

H. Pople: "The Dialog Model of Diagnostic Logic and Its Use in Internal Medicine", Proceedings of the Fourth International Joint Conference on Artificial Inteiligence, 1975.

H. Pople: " The Formation of Composite Hypotheses in Diagnostic Problem Solving - An Exercise in Synthetic Reasoning", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp 1030-1037.

J. A. Robinson: "A Machine-Oriented Logic Based on the Resolution Principle", Journal of the Association for Computing Machinery, Vol. 12 No. 1, 1965, pp 23-41.

J. P. Roth, W. G. Bouricius, P. R. Schneider: "Programmed Algorithm to Compute Tests to Detect and Distinguish Faults in Logic Circuits", IEEE Transactions on Electronic Computers, Vol EC-16 No 5, October 1967.

E. Shortliffe: *MYCIN: Computer-Based Medical Consultation*, American Elsevier, 1976.

D. E. Smith, M. R. Genesereth: "Ordering Conjuncts in Problem Solving", HPP-82-9, Stanford University Heuristic Programming Project, March 1983.

D. E. Smith, M. R. Genesereth: "Finding All of the Solutions to a Problem", HPP-83-21, Stanford University Heuristic Programming Project, April 1983.

G. J. Sussman, G. L. Steele: "Constraints - A Language for Expressing Almost-Hierarchical Descriptions" *Artificial Intelligence* Vol 14, 1980, pp 1-39.

M. Stefik: "Planning with Constraints", *Artificial Intelligence* Vol 16, 1981, pp 111-140.

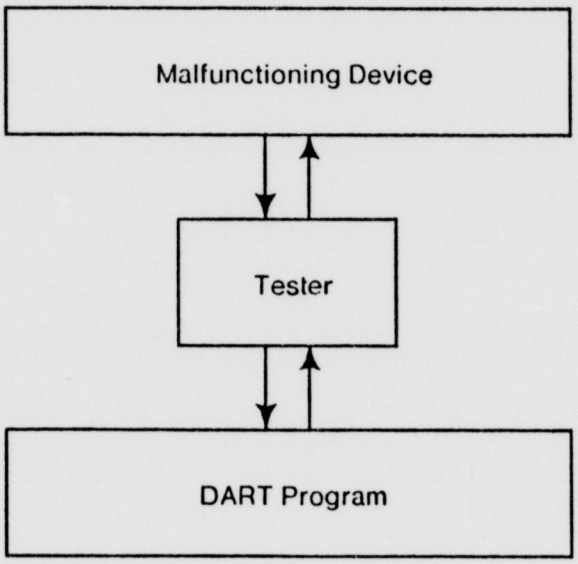

Figure 1 - Automated diagnosis of equipment failures

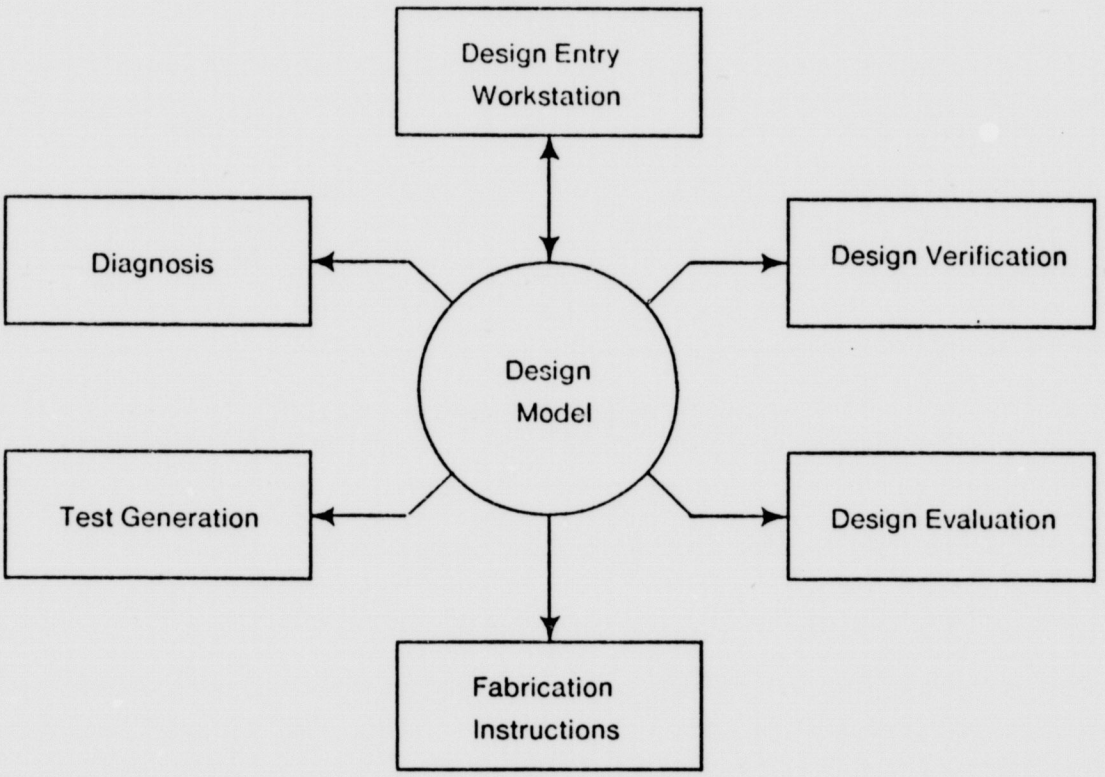

Figure 2 · An integrated design environment

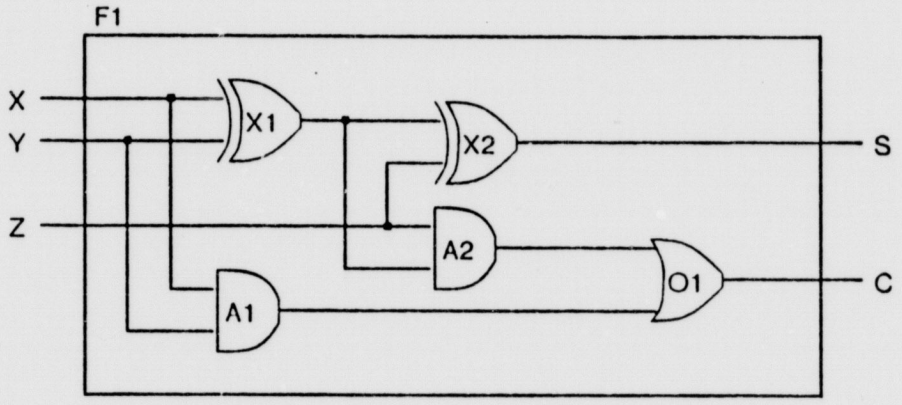

Figure 3 - Design of the full-adder F1

X	Y	Z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 4 - Behavior of the full adder F1

"And" Gates		
0	0	0
0	1	0
1	0	0
1	1	1

"Or" Gates		
0	0	0
0	1	1
1	0	1
1	1	1

"Xor" Gates		
0	0	0
0	1	1
1	0	1
1	1	0

Connections	
0	0
1	1

Figure 5 - Behavior of various gates

```
SD1:  (XORG X1)
SD2:  (XORG X2)
SD3:  (ANDG A1)
SD4:  (ANDG A2)
SD5:  (ORG O1)

SD6:  (CONN (IN 1 F1) (IN 1 X1))
SD7:  (CONN (IN 1 F1) (IN 1 A1))
SD8:  (CONN (IN 2 F1) (IN 2 X1))
SD9:  (CONN (IN 2 F1) (IN 2 A1))
SD10: (CONN (IN 3 F1) (IN 2 X2))
SD11: (CONN (IN 3 F1) (IN 1 A2))
SD12: (CONN (OUT 1 X1) (IN 1 X2))
SD13: (CONN (OUT 1 X1) (IN 2 A2))
SD14: (CONN (OUT 1 A1) (IN 2 O1))
SD15: (CONN (OUT 1 A2) (IN 1 O1))
SD16: (CONN (OUT 1 X2) (OUT 1 F1))
SD17: (CONN (OUT 1 O1) (OUT 2 F1))
```

Figure 6 - Structural description for the full adder F1

```
BD1: (IF (AND (VAL (IN 1 F1) t OFF) (VAL (IN 2 F1) t OFF) (VAL (IN 3 F1) t OFF))
         (AND (VAL (OUT 1 F1) t OFF) (VAL (OUT 2 F1) t OFF)))

BD2: (IF (AND (VAL (IN 1 F1) t OFF) (VAL (IN 2 F1) t OFF) (VAL (IN 3 F1) t ON))
         (AND (VAL (OUT 1 F1) t ON) (VAL (OUT 2 F1) t OFF)))

BD3: (IF (AND (VAL (IN 1 F1) t OFF) (VAL (IN 2 F1) t ON)  (VAL (IN 3 F1) t OFF))
         (AND (VAL (OUT 1 F1) t ON) (VAL (OUT 2 F1) t OFF)))

BD4: (IF (AND (VAL (IN 1 F1) t OFF) (VAL (IN 2 F1) t ON)  (VAL (IN 3 F1) t ON))
         (AND (VAL (OUT 1 F1) t OFF) (VAL (OUT 2 F1) t ON)))

BD5: (IF (AND (VAL (IN 1 F1) t ON)  (VAL (IN 2 F1) t OFF) (VAL (IN 3 F1) t OFF))
         (AND (VAL (OUT 1 F1) t CN) (VAL (OUT 2 F1) t OFF)))

BD6: (IF (AND (VAL (IN 1 F1) t ON)  (VAL (IN 2 F1) t OFF) (VAL (IN 3 F1) t ON))
         (AND (VAL (OUT 1 F1) t OFF) (VAL (OUT 2 F1) t ON)))

BD7: (IF (AND (VAL (IN 1 F1) t ON)  (VAL (IN 2 F1) t ON)  (VAL (IN 3 F1) t OFF))
         (AND (VAL (OUT 1 F1) t OFF) (VAL (OUT 2 F1) t ON)))

BD8: (IF (AND (VAL (IN 1 F1) t ON)  (VAL (IN 2 F1) t ON)  (VAL (IN 3 F1) t ON))
         (AND (VAL (OUT 1 F1) t ON) (VAL (OUT 2 F1) t ON)))
```

Figure 7 - Behavioral description for the full adder F1

```
TH1:  (IF (AND (ANDG d) (VAL (IN 1 d) t ON) (VAL (IN 2 d) t ON))
          (VAL (OUT 1 d) t ON))
TH2:  (IF (AND (ANDG d) (VAL (IN 1 d) t OFF))
          (VAL (OUT 1 d) t OFF))
TH3:  (IF (AND (ANDG d) (VAL (IN 2 d) t OFF))
          (VAL (OUT 1 d) t OFF))

TH4:  (IF (AND (ORG d) (VAL (IN 1 d) t OFF) (VAL (IN 2 d) t OFF))
          (VAL (OUT 1 d) t OFF))
TH5:  (IF (AND (ORG d) (VAL (IN 1 d) t ON))
          (VAL (OUT 1 d) t ON))
TH6:  (IF (AND (ORG d) (VAL (IN 2 d) t ON))
          (VAL (OUT 1 d) t ON))

TH7:  (IF (AND (XORG d) (VAL (IN 1 d) t ON) (VAL (IN 2 d) t ON))
          (VAL (OUT 1 d) t OFF))
TH8:  (IF (AND (XORG d) (VAL (IN 1 d) t ON) (VAL (IN 2 d) t OFF))
          (VAL (OUT 1 d) t ON))
TH9:  (IF (AND (XORG d) (VAL (IN 1 d) t OFF) (VAL (IN 2 d) t ON))
          (VAL (OUT 1 d) t ON))
TH10: (IF (AND (XORG d) (VAL (IN 1 d) t OFF) (VAL (IN 2 d) t OFF))
          (VAL (OUT 1 d) t OFF))

TH11: (IF (AND (CONN x y) (VAL x t z))
          (VAL y t z))
```

Figure 8 - Behavioral description of logic gates and connections

```
AC1: (VAL (IN 1 F1) 1 ON)
AC2: (VAL (IN 2 F1) 1 OFF)
AC3: (VAL (IN 3 F1) 1 OFF)

AC4: (VAL (IN 1 F1) 2 ON)
AC5: (VAL (IN 2 F1) 2 OFF)
AC6: (VAL (IN 3 F1) 2 ON)
```

Figure 9 - Achievable data concerning the full adder F1

```
OB1: (VAL (OUT 1 F1) 1 OFF)
OB2: (VAL (OUT 2 F1) 1 OFF)

OB3: (VAL (OUT 2 F1) 2 OFF)
```

Figure 10 - Observable data concerning the full adder F1

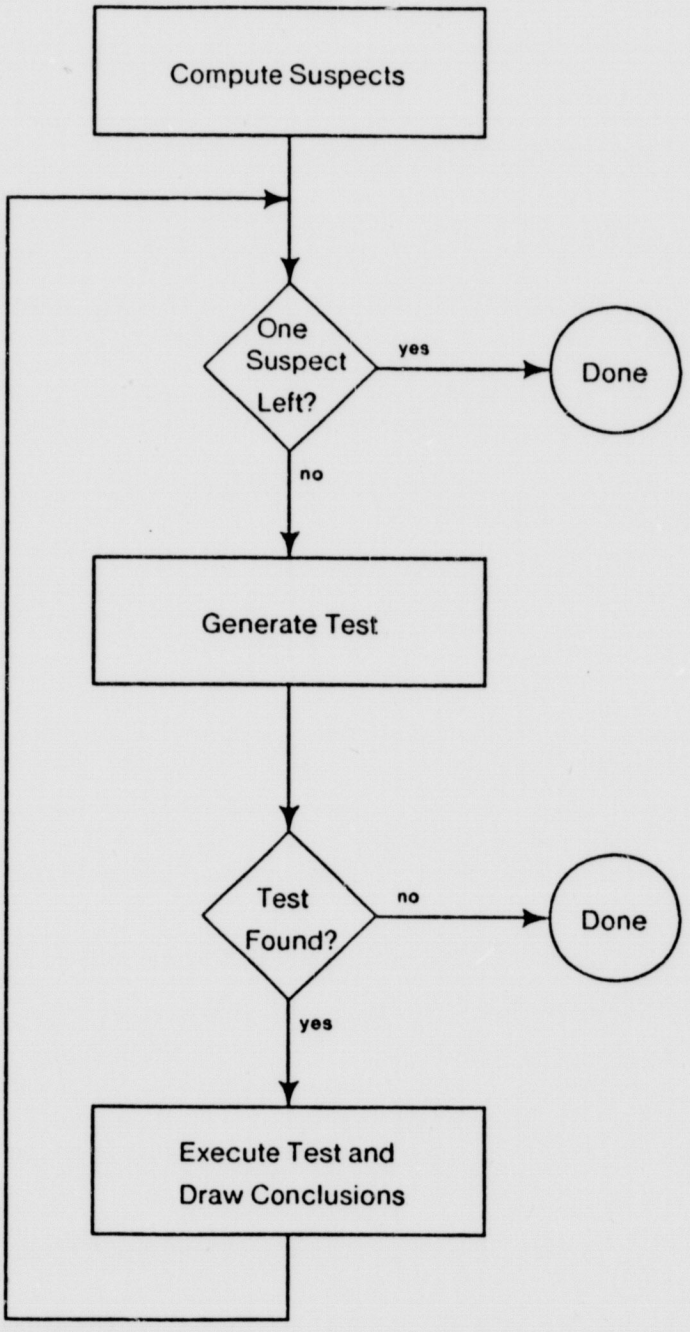

Figure 12 - Flowchart for the DART Program

```
DA1:  (CONN (IN 1 F1) (IN 1 X1))
DA2:  (CONN (IN 1 F1) (IN 1 A1))
DA3:  (CONN (IN 2 F1) (IN 2 X1))
DA4:  (CONN (IN 2 F1) (IN 2 A1))
DA5:  (CONN (IN 3 F1) (IN 2 X2))
DA6:  (CONN (IN 3 F1) (IN 1 A2))
DA7:  (CONN (OUT 1 X1) (IN 1 X2))
DA8:  (CONN (OUT 1 X1) (IN 2 A2))
DA9:  (CONN (OUT 1 A1) (IN 2 O1))
DA10: (CONN (OUT 1 A2) (IN 1 O1))
DA11: (CONN (OUT 1 X2) (OUT 1 F1))
DA12: (CONN (OUT 1 O1) (OUT 2 F1))

DA13: (IF (NOT (XORG X1)) (AND (XORG X2) (ANDG A1) (ANDG A2) (ORG O1)))
DA14: (IF (NOT (XORG X2)) (AND (XORG X1) (ANDG A1) (ANDG A2) (ORG O1)))
DA15: (IF (NOT (ANDG A1)) (AND (XORG X1) (XORG X2) (ANDG A2) (ORG O1)))
DA16: (IF (NOT (ANDG A2)) (AND (XORG X1) (XORG X2) (ANDG A1) (ORG O1)))
DA17: (IF (NOT (ORG O1))  (AND (XORG X1) (XORG X2) (ANDG A1) (ANDG A2)))

DA18: (IF (AND (VAL (IN 1 d) s x) (VAL (IN 2 d) s y) (VAL (OUT 1 d) s z)
               (VAL (IN 1 d) t x) (VAL (IN 2 d) t y))
          (VAL (OUT 1 d) t z))
```

Figure 11 - Device Assumptions for the full adder F1

```
CS1:  (NOT (VAL (OUT 1 F1) 1 ON))                                   OB1

CS2:  (OR (CONN x (OUT 1 F1))                                       TH11
          (NOT (VAL x 1 ON)))

CS3:  (NOT (VAL (OUT 1 X2) 1 ON))                                   DA11

CS4:  (OR (NOT (XORG X2))                                           TH8
          (NOT (VAL (IN 1 X2) 1 ON))
          (NOT (VAL (IN 2 X2) 1 OFF)))

CS5:  (OR (NOT (XORG X2))                                           TH11  DA7
          (NOT (VAL (OUT 1 X1) 1 ON))
          (NOT (VAL (IN 2 X2) 1 OFF)))

CS6:  (OR (NOT (XORG X2))                                           TH8
          (NOT (XORG X1))
          (NOT (VAL (IN 1 X1) 1 ON))
          (NOT (VAL (IN 2 X1) 1 OFF))
          (NOT (VAL (IN 2 X2) 1 OFF)))

CS7:  (OR (NOT (XORG X2))                                           TH11  DA1
          (NOT (XORG X1))
          (NOT (VAL (IN 1 F1) 1 ON))
          (NOT (VAL (IN 2 X1) 1 OFF))
          (NOT (VAL (IN 2 X2) 1 OFF)))

CS8:  (OR (NOT (XORG X2))                                           AC1
          (NOT (XORG X1))
          (NOT (VAL (IN 2 X1) 1 OFF))
          (NOT (VAL (IN 2 X2) 1 OFF)))

CS9:  (OR (NOT (XORG X2))                                           TH11  DA3
          (NOT (XORG X1))
          (NOT (VAL (IN 2 F1) 1 OFF))
          (NOT (VAL (IN 2 X2) 1 OFF)))

CS10: (OR (NOT (XORG X2))                                           AC2
          (NOT (XORG X1))
          (NOT (VAL (IN 2 X2) 1 OFF)))

CS11: (OR (NOT (XORG X2))                                           TH11  DA5
          (NOT (XORG X1))
          (NOT (VAL (IN 3 F1) 1 OFF)))

CS12: (OR (NOT (XORG X2))                                           AC3
          (NOT (XORG X1)))

CS13: (ANDG A1)                                                     DA13  DA14

CS14: (ANDG A2)                                                     DA13  DA14

CS15: (ORG O1)                                                      DA13  DA14
```

Figure 13 - Example of computing suspects

```
GT1:  (OR (NOT (XORG X1)) (XORG X1))

GT2:  (OR (NOT (XORG X1))                                          TH8
          (NOT (VAL (IN 1 X1) t ON))
          (NOT (VAL (IN 2 X1) t OFF))
          (VAL (OUT 1 X1) t ON))

GT3:  (OR (NOT (XORG X1))                                          TH11  DA1
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 X1) t OFF))
          (VAL (OUT 1 X1) t ON))

GT4:  (OR (NOT (XORG X1))                                          TH11  DA3
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (VAL (OUT 1 X1) t ON))

GT5:  (OR (NOT (XORG X1))                                          TH11  DA8
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (VAL (IN 2 A2) t ON))

GT6:  (OR (NOT (XORG X1))                                          TH1
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (NOT (ANDG A2))
          (NOT (VAL (IN 1 A2) t ON))
          (VAL (OUT 1 A2) t ON))

GT7:  (OR (NOT (XORG X1))                                          CS14
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (NOT (VAL (IN 1 A2) t ON))
          (VAL (OUT 1 A2) t ON))

GT8:  (OR (NOT (XORG X1))                                          TH11  DA6
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (NOT (VAL (IN 3 F1) t ON))
          (VAL (OUT 1 A2) t ON))

GT9:  (OR (NOT (XORG X1))                                          TH11  DA10
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (NOT (VAL (IN 3 F1) t ON))
          (VAL (IN 1 O1) t ON))

GT10: (OR (NOT (XORG X1))                                          TH5
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (NOT (VAL (IN 3 F1) t ON))
          (NOT (ORG O1))
          (VAL (OUT 1 O1) t ON))

GT11: (OR (NOT (XORG X1))                                          CS15
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (NOT (VAL (IN 3 F1) t ON))
```

```
          (VAL (OUT 1 O1) t ON))

GT12: (OR (NOT (XORG X1))                                    TH11  DA12
          (NOT (VAL (IN 1 F1) t ON))
          (NOT (VAL (IN 2 F1) t OFF))
          (NOT (VAL (IN 3 F1) t ON))
          (VAL (OUT 2 F1) t ON))
```

Figure 14 - Example of generating tests

```
DC1:  (NOT (VAL (OUT 2 F1) 2 ON))                                  OB3

DC2:  (OR (NOT (XORG X1))                                          GT12
          (NOT (VAL (IN 1 F1) 2 ON)
          (NOT (VAL (IN 2 F1) 2 OFF))
          (NOT (VAL (IN 3 F1) 2 ON)))

DC3:  (OR (NOT (XORG X1))                                          AC4
          (NOT (VAL (IN 2 F1) 2 OFF))
          (NOT (VAL (IN 3 F1) 2 ON)))

DC4:  (OR (NOT (XORG X1))                                          AC5
          (NOT (VAL (IN 3 F1) 2 ON)))

DC5:  (NOT (XORG X1))                                              AC6
```

Figure 15 - Example of drawing conclusions from tests

Copyright © 1985 by HPP and
Comtex Scientific Corporation

FILMED FROM BEST AVAILABLE COPY