Diagnosis Using Hierarchical Design Models

Michael R. Genesereth
Stanford University
Stanford, California 94305

Abstract: This paper presents a new algorithm for the diagnosis of computer hardware faults. The algorithm uses a general inference procedure to compute suspect components and generate discriminatory tests from information about the design of the device being diagnosed. In the current implementation this procedure is linear-input resolution, guided by explicit meta-level control rules. The algorithm exploits the hierarchy inherent in most computer system designs to diagnose systems a level at a time. In this way the number of parts under consideration at any one time is kept small, and the cost of test generation remains manageable.

1. Introduction

Being physical devices, computer components are subject to failure; and, given current design practices, the failure of one component can lead to the malfunction of an entire computer system. This paper describes an automated diagnostician for computer hardware faults. The program accepts a statement of a system malfunction in a formal language, suggests tests and accepts the results, and ultimately pinpoints the components responsible for the failure. In recognition of its intended role as an assistant for humam field engineers, the program is called DART for Diagnostic Assistance Reference Tool.

The DART program was developed in the context of moderately successful work on medical diagnosis, as exemplified by such programs as CASNET [Weiss, Kulikowski], INTERNIST [Pople 1975] [Pople 1977], and MYCIN [Shortliffe]. However, there's a difference. The medical diagnosis programs all utilize "rules" that associate symptoms with possible diseases. The DART program contains no information about how computers fail. Instead, it works directly from information about *intended* structure (a machine's parts and their interconnections) and *expected* behavior (equations. rules, or procedures that relate inputs and outputs).

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, then when he is done his design will be online. The structural and behavioral information in this model can then be passed as data to the DART program to diagnose its faults. Similarly, the design information could be passed to a program to generate manufacturing instructions and to another program to generate testing codes.

This work was done with the support and collaboration of the Palo Alto Scientific Center and the Field Engineering Division of IBM.

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, the most well-known of which is the d-algorithm [Roth et al]. Its primary disadvantage is its runtime. While 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], the cost of generating appropriate tests grows polynomially or exponentially. (In fact, the problem is NP-complete [Sahni and Ibarra].) Since the d-algorithm works only at the gate level, it is impractical for circuits the size of current computer systems.

The DART program meets this difficulty by exploiting the hierarchy inherent in most computer system designs. (See figure 1.) The program first diagnoses the system at a high level of abstraction to determine the major subcomponent in which the fault lies (e.g. the adder A1). It then focusses its attention on the next lower level (e.g. finding the full-adder F2) and then repeats until it can identify a replaceable part (e.g. the xor gate R3). In this way, the number of components under consideration at any one time is kept small, and the cost of test generation remains manageable.

Within each level DART uses a deductive procedure to compute suspects and generate tests. All symptoms are expressed as violations of expected behavior. Starting with a symptom of this sort, DART reasons backwards from the expected behavior to discover why it was expected and in so doing produces a justification for its conclusions. Since the expected behavior was not observed, all of the parts mentioned in this justification are suspect. The next step is to generate a test to discriminate among these suspects. DART starts with a behavioral rule for one of the suspects and works forward to observable outputs and backwards to modifiable inputs. The result of this step is an expectation for certain outputs when certain inputs are applied. If the outputs do not have the values expected, then one of the parts mentioned in the derivation of the test must be broken.

Section 2 presents a simplified version of the design description language used by DART, and section 3

describes the diagnostic procedure itself. Note that, although the examples in this paper are all time-independent, the language is equally versatile at describing time-dependent behavior, and the DART algorithm generalizes as well. Section 4 offers an analysis of the completeness and efficiency of the algorithm and pinpoints some of its shortcomings. The conclusion discusses the state of implementation and testing and summarizes the key points of the paper. This paper is an abridged version of an earlier paper [Genesereth].

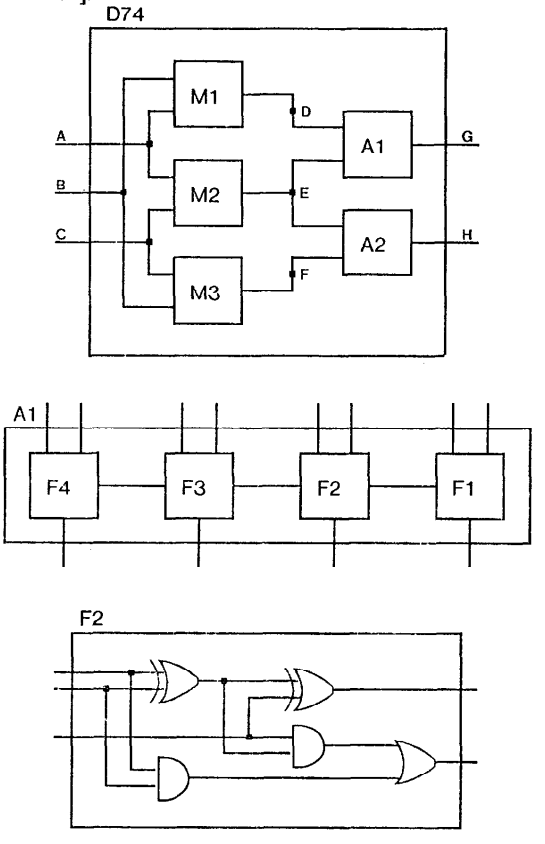

Figure 1 - A Hierarchical Design

2. Design Models

The DART procedure expects as data a full description of the structure and behavior of the circuit to be diagnosed. This information must be in the form of data base assertions in a design description language called SUBTLE. This language has the advantage of being equally expressive at all levels of description; and, importantly, it is adequate for encoding tests and assumptions about the circuit, e.g. the single fault assumption or nonintermittency.

The syntax of SUBTLE is that same as that of predicate calculus, and the rules should be apparent after a few examples. The vocabulary is described in the following sections. Three syntactic conventions are used in this paper to simplify the examples. Upper case letters are used exclusively for constants, functions, and relations, while lower case letters are used for variables. Prefix universal quantifiers are dropped, and all free variables can be assumed to be universally quantified. Finally, for the sake of brevity underlining is used to denote negation.

2.1 Structural Vocabulary

The structure of a device is specified by describing its parts and their interconnections. The structure of each part can in turn be described until one reaches one's "primitive" components (which are usually characterized behaviorally).

In SUBTLE each part is designated by an atomic name (e.g. `A1`). The type of each part is declared using type relations. The following assertions declare the types of the toplevel components of the circuit in figure 1: `M1`, `M2`, and `M3` are multipliers; `A1` and `A2` are adders.

```
MULT(M1)
MULT(M2)
MULT(M3)
ADDER(A1)
ADDER(A2)
```

Every device in SUBTLE has zero or more inputs and outputs, and these "ports" are designated using the functions `INi` and `OUTi`. For example, `IN2(M1)` would designate the second input of `M1`, and `OUT2(FA1)` would designate the second output of `FA1`. If a device has only one input or output, the trailing digit is omitted. Connections are made between the ports of devices. The next 4 assertions specify the wiring diagram for `D74`. For example, the first assertion states that the output of `M1` is connected to the first input of `A1`.

```
CONN(OUT(M1), IN1(A1))
CONN(OUT(M2), IN2(A1))
CONN(OUT(M2), IN1(A2))
CONN(OUT(M3), IN2(A2))
```

2.2 Behavioral Vocabulary

The behavior of a circuit can usually be expressed in terms of the signal values at the circuit's inputs and outputs. In SUBTLE, the signal at an input or output at a given time is specified by superscripting the input or output function and equating it to its value. For example, the first assertion below states that at time `1`, the second input of `A1` is `3`. The second assertion states that the output of `M1` is always `1`.

$$IN2^{1}(A1)=3$$
$$OUT^{t}(M1)=1$$

The simplest form of behavioral specification is a set of rules relating a circuit's inputs and outputs. For example, the behavior of an adder can be captured by the following rules. `OK(p)` is intended to mean that `p` is operational, i.e. not broken.

$$ADDER(r) \wedge OK(r) \wedge IN1^{t}(r)=x \wedge IN2^{t}(r)=y \rightarrow OUT1^{t}(r)=x+y$$
$$ADDER(r) \wedge OK(r) \wedge IN1^{t}(r)=x \wedge OUT1^{t}(r)=z \rightarrow IN2^{t}(r)=z-x$$
$$ADDER(r) \wedge OK(r) \wedge OUT1^{t}(r)=z \wedge IN2^{t}(r)=y \rightarrow IN1^{t}(r)=z-y$$

As with structural information, behavioral descriptions are frequently hierarchical, with signals being characterized differently at one level of the structural hierarchy than at another. In order for DART to diagnose a circuit, it must have a formal statement of the relationship between the signals at these different levels. A good way of encoding this information is in the form of propositions relating the inputs and outputs of a device at one level with their counterparts at the next lower level. For example, the following propositions capture the mapping between the integers manipulated by an adder and the bits manipulated by its subparts.

$$IN1^t(A1) = IN1^t(F4)*8 + IN1^t(F3)*4 + IN1^t(F2)*2 + IN1^t(F1)*1$$
$$IN2^t(A1) = IN2^t(F4)*8 + IN2^t(F3)*4 + IN2^t(F2)*2 + IN2^t(F1)*1$$
$$OUT^t(A1) = OUT1^t(F4)*8 + OUT1^t(F3)*4 + OUT2^t(F2)*2 + OUT2^t(F1)*1$$

Finally, there is a set of behavioral rules for connections. The following proposition states that, if two terminals are connected, they always bear the same signal.

$$CONN(OUT(r), IN(s)) \rightarrow OUT^t(r) = IN^t(s)$$

2.3 Fault Assumptions

Two assumptions that are important to the efficiency and power of test generation procedures are the single fault assumption and nonintermittency. Neither is essential for the operation of the DART algorithm. However, by adding formal statements of these assumptions to DART's global data base, one can get the algorithm to take advantage of them.

The single fault assumption states that at most one component in a circuit is faulty. This idea can be formalized by stating that the non-functionality of any component implies the functionality of all the rest. For the top-level parts of the device in figure 1, this would lead to the following propositions.

$$\underline{OK(M1)} \rightarrow OK(M2) \wedge OK(M3) \wedge OK(A1) \wedge OK(A2)$$
$$\underline{OK(M2)} \rightarrow OK(M1) \wedge OK(M3) \wedge OK(A1) \wedge OK(A2)$$
$$\underline{OK(M3)} \rightarrow OK(M1) \wedge OK(M2) \wedge OK(A1) \wedge OK(A2)$$
$$\underline{OK(A1)} \rightarrow OK(M1) \wedge OK(M2) \wedge OK(M3) \wedge OK(A2)$$
$$\underline{OK(A2)} \rightarrow OK(M1) \wedge OK(M2) \wedge OK(M3) \wedge OK(A1)$$

The nonintermittency assumption 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, and it is essential in the derivation of some powerful tests. The proposition below encodes the assumption formally. It states that, if a device with sepcific inputs has a specific output at one time, then given the same inputs it will have the same output at any other time.

$$IN1^s(r)=x \wedge IN2^s(r)=y \wedge OUT^s(r)=z \wedge IN^t(r)=x \wedge IN2^t(r)=y \rightarrow OUT^t(r)=z$$

3. The DART Algorithm

The DART procedure begins with the design description for the device under test and a set of observed symptoms. It produces as output the minimal set of replaceable parts that will correct the error. Especially useful to the procedure are the assumptions that the fault occurs within a single replaceable part and is not intermittent. While neither assumption is logically necessary, the performance of the algorithm degrades without them.

Figure 2 presents an overall flowchart. The first step is to diagnose the fault at the highest level in the structural hierarchy. This may result in more than one part if the problem is not diagnosable at that level. If the implicated parts are replaceable, the diagnosis is complete. Otherwise, DART examines them at the next lower level of detail.

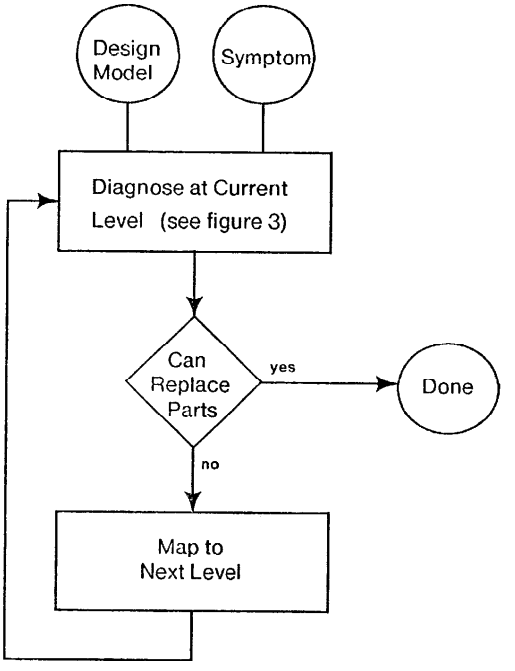

Figure 2 - Overall View of DART Algorithm

The algorithm treats each level of the hierarchy in the same way. Figure 3 presents a flowchart for the diagnostic component. The algorithm first uses information about the symptoms to compute a list of suspect parts. If this list contains only a single element, the diagnosis is complete. Otherwise, the next step is to devise a test to discriminate the suspects. The test is then executed, and the process repeats. If no further discriminatory tests can be found, the problem is undiagnosable and a list of the remaining suspects is returned as value.

The workhorse of the DART algorithm is a general inference procedure. In the current implementation this procedure is linear-input resolution, guided by a set of explicit meta-level control rules.

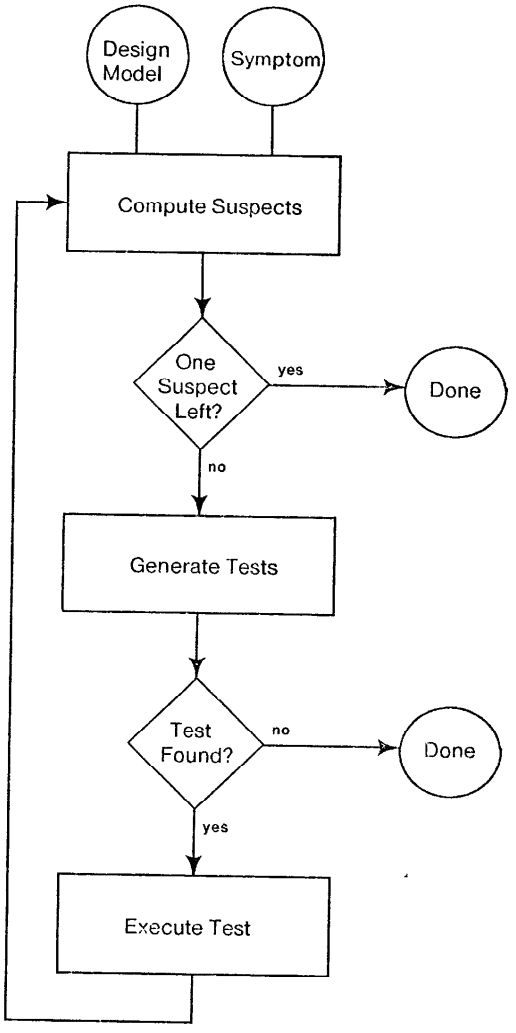

Figure 3 - DART Algorithm at Each Level

3.1 Computing Suspects

The goal of suspect computation is a proposition of the following form, where each Pi is a statement about the structure of the circuit, e.g. OK(A1) or CONN(OUT(M1), IN1(A1)).

P1 v . . . v Pn

In generating suspects DART starts with a symptom, i.e. the negation of an expectation, and resolves it with the rules in the device's design model until it produces a proposition of this form.

As an example, consider the circuit shown in figure 1. Assume that for inputs A=1, B=1, and C=3 the device produces 2 as the value of output G instead of the correct result 4. Since G=4 and ADDER(A1), DART can conclude (D=1 v E=3 v OK(A1)). The only way that D=1 could be true is if (A=1 v B=1 v OK(M1)), and the only way that E=1 could be true is if (A=1 v C=3 v OK(A1)). Since the inputs A, B, and C are known, the system can conclude the following suspect proposition.

OK(M1) v OK(M2) v OK(A1)

One thing to note about this technique is that it is equivalent to proving the expected value and then looking at the dependencies for all propositions of the form OK(p) or CONN(x,y).

Another thing to note is that suspect computation is not simply a matter of tracing the circuit diagram backwards from a faulty output. On the one hand, this can lead to too many suspects. For example, if the output of a multiplier is expected to be zero because its second input is zero, then in computing suspects for a nonzero output, it isn't necessary to look at the predecessors of the first input. 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.

3.2 Generating Tests

The goal of DART's test generation procedure is a proposition of the following form, where each of the Ii is a modifiable input, each of the pi is an immediate subcomponent of the device being diagnosed and O is an observable output. In order to have discriminatory power, the pi should not include all of the current suspects.

OK(p1) & . . & OK(pn) & I1 & . . . & In => O

In generating tests, DART starts with a behavioral rule for one of the suspects and generates conclusions using the rules from the device's design model until a proposition of the appropriate form is generated.

As an example, consider the generation of a test to discriminate the suspects in the example introduced in the last section. Starting with a rule for the multiplier M2, DART is able to produce a test with the same inputs.

OK(M2) & OK(M3) & OK(A2) => H=6

If H=6, then the fault must lie in one of M2, M3, or A2. M3 and A2 are not mentioned in the previous suspect list; and so, under a single fault assumption, they are exonerated; and the fault must lie in M2. If H=6, M2 can be exonerated by a little deduction. If M2 is responsible for the erroneous value on output G, it must be because its ouput is wrong. Therefore, if M3 and A2 are okay, H must also be wrong. Since H=6 as expected, M2 must be okay.

As another example consider the problem of discriminating the remaining two suspects M1 and A1. DART reasons as follows. If the fault lies in M1, then A1 is ok; and, for G to be 2, D must be -1. Therefore, if E is changed to 4, G must change to 3. If it doesn't, then A1 is broken, and M1 can be exonerated. (Note that, if a 4 appears as predicted, this does not exonerate A1.) Next the program figures out that, in order to get a 4 on line E

without disturbing the inputs to M1, input C must be changed to 4. As a proposition this test can be represented as follows.

```
OK(A1) & A=1 & B=1 & C=4 => G=3
```

In this case suppose that G changes to 5. This is correct but not the predicted result. In light of the previous symptom, it demonstrates that A1 is faulty.

3.3 Control

Each of the steps in the DART algorithm uses the resolution rule to derive new conclusions until appropriate termination criteria are met. For example, in generating tests DART begins with a behavioral rule and draws conclusions using the facts from the circuit's design model. From these conclusions, it draws other conclusions and so on until a "test" is derived.

If the resolution rule were applied without guidance, it would produce numerous useless propositions. To mitigate this problem, DART draws only those conclusions in which at least one of the resolvents is an assertion in the design model. No resolutions are performed between conclusions and other conclusions. This restriction is often called the "linear input strategy" [Nilsson] and is very effective in preventing an undesirable proliferation of useless conclusions.

DART orders its deductions by the size of the conclusion relative to the sizes of the resolvents, with the smallest being processed first. A special case of this occurs when one of the resolvents is an atomic proposition, in which case the conclusion has one fewer disjuncts than the other resolvent. This is often called the "unit preference strategy" [Nilsson].

DART also postpones the instantiation of variables as long as possible. This is equivalent to the constraint propagation technique used in the revised d-algorithm and described more recently in the AI literature [Sussman, Steele] [Stefik].

4. Analysis of the Algorithm

The two key factors in analyzing a diagnostic procedure are its diagnostic completeness and computational efficiency.

4.1 Completeness

The DART algorithm is almost complete in that it is can generate most tests of this form; and a slight variant (ancestry-filtered resolution), though computationally more expensive, is guaranteed to generate every test. Even so, due to the limited vocabulary at a given level of description, the set of tests so defined is sometimes inadequate to localize every fault.

4.2 Efficiency

The problem of test generation has been shown to be NP-complete [Ibarra and Sahni], and so in the worst case the runtime of a complete test generation procedure is likely to be exponential in the number of subparts involved. In the worst case, the DART algorithm is no better than any other procedure. However, for many circuits the cost of diagnosis as a function of the number of components is approximately the same at each level of description; and, when this is the case, hierarchical diagnosis leads to substantial computational savings.

Assuming that at each level the cost of diagnosis is some function F_i and the number of gates 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 gates 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 best case of a linear cost, the hierarchical approach still offers a logarithmic advantage.

5. Conclusions

The DART algorithm has been implemented and tested on a variety of examples. The implementation was done in COMMON LISP [Steele and Fahlman] with the help of a data base and inference system called MRS [Genesereth, Greiner, Smith]. The examples include simple circuits like the one in the introduction as well as complex systems like the teleprocessing facility of the IBM 4331. In all cases the algorithm was able to generate appropriate tests and diagnose the underlying fault. Running on a VAX-11/780, the time to diagnose each case was on the order of minutes. While this is not particularly good for small circuits, it could be improved significantly by more efficient programming style. And, importantly, the program is able to diagnose large circuits without significant slowdown.

In summary, the key contributions of this research are (1) its demonstration of the value of hierarchical design models in diagnostic test generation and (2) the design of the DART algorithm. DART utilizes the hierarchy inherent in most computer systems to keep the number of parts under consideration small and thereby reaps substantial computational savings. Furthermore, since the method works directly with design models, rules associating symptoms with specific failures are unnecessary. For these reasons, the algorithm appears to be a promising way of coping with the increasing complexity of modern computer system designs.

References

J. S. Bennett and C. R. Hollander, Dart, Proceedings of the 7th International Joint Conference on Artificial Intelligence, August 1981.

M. A. Breuer, M. Abramovici: "Fault Diagnosis Based on Effect-Cause Analysis", 17th Design Automation Conference Proceedings, June 1980.

M. R. Genesereth, R. Greiner, D. E. Smith: "MRS Manual", HPP-81-6, Stanford University Heuristic Programming Project, December 1981.

M. R. Genesereth, M. Grinberg, J. Lark", HPP-81-11, Stanford University Heuristic Programming Project, September 1981.

M. R. Genesereth: "The Use of Hierarchical Design Models in the Automated Diagnosis of Computer Hardware Faults", HPP-81-20, Stanford University Heuristic Programming Project, December 1981.

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.

N. Nilsson: *Principles of Artificial Intelligence*, Tioga Press, 1980.

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

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

G. L. Steele, S. E. Fahlman: "COMMON LISP Manual", SPICE Project, Carnegie-Mellon University, September 1981.

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

S. Weiss, C. Kulikowski, A. Safir: "A Model-Based Consultation System for the Long-Term Management of Glaucoma, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp 826-832.