AN APPROACH TO DEPENDENCY DIRECTED BACKTRACKING USING DOMAIN SPECIFIC KNOWLEDGE

Vasant Dhar
Graduate School of Business Administration
New York University

Casey Quayle
Bolt, Beranek and Newman
Boston, Massachussetts

April 1985

Center for Research on Information Systems
Computer Applications and Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #89

GBA #85-21(CR)

A shorter version of this paper appears in the Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI), 1985.

Abstract

The idea of dependency directed backtracking proposed by Stallman and Sussman (1977) offers significant advantages over heuristic search schemes with chronological backtracking which waste much effort by discarding many "good" choices when backtracking situations arise. However, we have found that existing non-chronological backtracking machinery is not suitable for certain types of problems, namely, those where choices do not *follow logically* from previous choices, but are based on a heuristic evaluation of a constrained set of alternatives. This is because a choice is not justified by a "set of support" (of previous choices), but because its advantages outweigh its drawbacks in comparison to its competitors. What is needed for these types of problems is a scheme where the advantages and disadvantages of choices are explicitly recorded during problem solving. Then, if an unacceptable situation arises, information about the nature of the unacceptability and the tradeoffs can be used to determine the most appropriate backtracking point. Further, this requires the problem solver to use its *hindsight* to preserve those "good" intervening choices that were made chronologically after the "bad" choice, and to resume its subsequent reasoning in light of the modified set of constraints. In this paper, we describe a problem solver for non-chronological backtracking in situations involving tradeoffs. By endowing the backtracker with access to domain-specific knowledge, a highly contextual approach to reasoning in dependency directed backtracking situations can be achieved.

Key Words: Dependency Directed Backtracking; Hindsight; Reasoning under Uncertainty; Context-based reasoning; Expert Systems; Domain Specific Reasoning

1. Introduction

An area of investigation now commonly referred to as "Dependency Directed Reasoning," which has resulted from the work of Stallman and Sussman (1977) and others, has had a major impact on AI research in the last decade. The basic ideas proposed by Stallman and Sussman have been extended by Doyle (1978, 1980) and others (de Kleer et. al, 1977; McDermott and Doyle, 1980; McAllester, 1982) leading to systems like the "Truth Maintenance System" (TMS) (Doyle, 1978) and RUP (McAllester, 1982) which can be used to build models of common sense reasoning and plausible inference. An integrating theme running through this research is one of "self aware" or "introspective" problem solvers that are able to account for their actions by maintaining records of reasons for choices. This "dependency information" can be used by a program introspectively, to examine and revise its set of beliefs whenever necessary, and to provide a user with explanations or rationales for its existing set of beliefs.

In this paper we describe some dependency directed reasoning features of a problem solver called PLANET (Dhar, 1984) It has been designed to help planning managers in a large computer manufacturing company (referred to here as "CMC") with the formulation and investigation of models for allocation of resources such as manpower, space, and capital. Since the process of resource planning[1] involves making assumptions that are continually subject to revision, dependency information plays a crucial role in the maintenance and incremental change of planning models. What is of particular interest in this paper is a heuristic procedure for dependency directed backtracking that addresses one drawback of existing dependency frameworks -- determining what belief (set of assumptions) in an existing model to change whenever an undesirable state arises.

2. PLANET Architecture -- An Overview

It takes many years from the time the introduction of a new machine is planned, to the time that a stable production process is realized. Planning for manufacturing complex computer systems constantly involves making assumptions which are continually refined as the scenario firms up. These projections pertain to various types of decisions such as make versus buy, where the various components will be produced or purchased, decisions about assembly, storage, testing, etc. -- tasks to be accomplished within limited resources. However, given the interrelationships among assumptions and the frequency with which they change, even the most carefully crafted model can become unreliable. Yet, if the organization is

[1]Unless otherwise stated, the term "plan" is used to mean *business* (manufacturing or resource) plan, and not a plan as normally understood in AI. To avoid possible confusion over these terms, we use the generic term "model" instead of plan wherever appropriate.

to maintain an accurate picture of its resource requirements, it must make constant adjustments to the manufacturing model[2] in the face of changing reality. The design of PLANET has been motivated by these considerations. Specifically, the program *preserves* the various alternatives that were contributed by the participants involved in the model formulation process, and then uses this and other domain-specific knowledge in order to reason about changing assumptions.

The alternatives maintained by the program are represented at several levels of abstraction. For example, "module-check", a complex manufacturing operation, involves several diagnostic activities, each of which could be carried out using different diagnostic equipment, which in turn might be used in various ways. Specifically, the program knows about *areas* of the manufacturing process (such as module-check, kernel-integration, peripherals-integration etc.), *activities* involved in these areas, *methods* (for testing, assembly, etc.), and *modes of use* (of the various methods). The complete set of alternatives that are considered in the course of formulating the manufacturing model can be visualized as a hierarchy of choices.[3] Figure 1 shows a part of the hierarchy used by

[2] The manufacturing model uniquely determines the resource plan.

[3] Each of these alternatives is represented as a structured object in HOUSE Quayle (1982), an object oriented programming system similar in spirit to the FLAVORS package. For example, "MIF-Test" (which stands for "manufacturing faults induced test") is an instance of an object of the "activity type." Similarly, "L-20" and "I-24" are instances of the "method type."

PLANET. This includes chosen assumptions, as well as those previously passed over. Entities at the bottom end of the lines (except for lines enclosed within dashes) represent alternate ways of accomplishing the entity at the top end of the line. The dashed lines include the *set* of activities that *must* be performed in an area.

2.1. The Process of Model Formulation

The problem of formulating a resource planning model has two important features. First, as a planning model is formulated, that is, as assumptions about various parts of the task environment are made, choices in other parts of the environment are constrained: a process commonly referred to as "constraint propagation." In this way, the appropriate relationships among different parts of the model are realized. Secondly, there are usually resource requirement tradeoffs among the alternatives that can be made. For example, table 1 shows various resource requirements for the "S-test" activity depending on what methods (here testing devices) are used. In making choices among such alternatives, the program uses an evaluation function to choose the most "balanced" alternative in light of the organization's resource availability picture at the time. Because these choices are made successively using limited look-ahead[4] it leaves open the possibility that some resource

[4] It does not know how many choices still need to be made, i.e. how much of the model still needs to be crafted, and what the tradeoffs involved will be.

constraint will be violated, thereby forcing the problem solver to undo one or more of its previous choices.

Table 1

These tables indicate resource-requirements/tradeoffs for four decisions. The selected alternative in the first three is in bold type. The units of Labor are workers working, Capital is in millions of dollars, and Space is in thousands of square feet for floor space.

Table 1a.
Module-check: MIF-Test

		Resource	
	Labor	Capital	Space
Shorts/opens-tester	**1**	**3**	**2**
DL	3	0.5	3

Table 1b.
Module-check: S-Test

		Resource:	
	Labor	Capital	Space
QV	1	2	4
L-20	**2**	**6**	**2**
FC-33	1	3	3

Table 1c.
Kernel-integration: Insertion

		Resource	
	Labor	Capital	Space
P1	**2**	**4**	1
P2	2	2	2
Manual	5	0.5	3

Table 1d.
Peripherals-integration: Peripherals-Test

		Resource:	
	Labor	Capital	Space
SIM-Test	3	4	3
FA-Test	3	3	6

As an example, consider the situation in Figure 2a showing four decisions involving tradeoffs (reflected in tables 1a, 1b, 1c and 1d) that have been considered during the model formulation process. The choices shown in the tables were made when "saving space" was considered more important than saving capital. That is, when faced with a tradeoff between space and capital, space was favored over capital. With this "money is no object" attitude the program is now in trouble. It cannot chose either the SIM-tester or the FA-tester, as either choice would more than consume available capital. Clearly some previous selection must be undone to alleviate the problem and several maneuvers are available. Under such circumstances, the problem solver must be capable of reasoning about the most rational course of action rather than simply making a blind selection. In the following paragraphs, we present a formal treatment of PLANET's choice process, a discussion of some of the problems we have encountered in using existing utility packages to model this process, and our approach toward resolving these problems in situations where involving tradeoffs.

3. Approaches to Dependency Directed Reasoning

3.1. Background

The problem solving approach of PLANET is similar to MOLGEN's (Stefik, 1980) "constraint posting" where processing continues with existing constraints until a quiescent state is reached. When the program quiesces, but has not completely solved the problem, it creates a new constraint by "guessing" and restarts processing. The guess is based on a heuristic evaluation function that compares alternative choices. The cycle of compute-quiesce-guess continues until the problem is solved.

Since the evaluation of alternatives when quiescent is heuristic, inappropriate guesses can lead to an over-constrained[5] state. When over-constrained, the program must find and retract an existing choice that contributed to the unacceptable state of affairs. An alternative choice is then made as a replacement, and processing continues. Stefik called this the UNDO operation; we refer to it as **Second Guessing.**

PLANET differs from MOLGEN in the way guessing and second guessing are done. PLANET employs a dependency directed mechanism in the spirit described by Stallman and Susman (1977) The result is an improvement in the quality of the second guessing.

Whenever the program is quiescent and must resort to guessing we say that it is making a "*Forced Choice*" about some part of the manufacturing process. In such situations, several alternatives typically

[5]The term "over-constrained" can be interpreted in different ways. In the TMS framework, it is a *logical contradiction*, i.e. a situation where a statement and its negation are both believed. In resource planning, an over-constrained situation is one where some resource/throughput constraint is violated. See Simon (1979) for examples of problem solving with resource constraints.

exist as illustrated in Figure 1. We call each forced choice a **decision** level item, where the program can no longer wait for more information about other parts of the manufacturing process - it must decide. Each of the alternatives available in the context of a decision level item is called a **selection** level item. Note that for any given decision level item there exists a set of at least two selection level items.

For the purpose of exposition we will denote particular decision level items with a subscripted D and denote particular selection level items with a subscripted S. Sets of either will be denoted in italics, i.e. *D*, and *S*.

When the program is processing a decision level item it is guessing. The guess is to pick some element from a set of selection level items. When the guess is made, only heuristic estimates of resource tradeoffs are available to evaluate the merits of particular alternatives.

By the time the program finds itself in an over-constrained state considerable computation has been done in developing a more accurate assessment of resource consumption and establishing other constraints. This information is not available to PLANET when guessing but is available when second guessing. The key point here is that the evaluation function used by the second guesser is more powerful than the evaluation function used by the guesser, allowing the program to continue problem solving with the benefit of its new hindsight. We will now describe how

PLANET employs this 20/20 hindsight.

When the program becomes over-constrained it has processed several decision level items. We call the set of these items $\hat{D}$. For each $D_i \epsilon \hat{D}$ there is a set, S_i of alternative selection level items (choice) that could have been guessed. We call the distinguished guess in S_i as S_{Ci}. Thus, the set of all selection level choice (guesses) that have been made is then $\hat{S}=\left\{S_{ci}\right\}$, for all i.

A subset, S, of $\hat{S}$ contains all S guesses that have contributed to the over-constrained state. S can be computed by chasing current dependency information. The subset, D, of $\hat{D}$ - the set of decision level choice points that need to be reconsidered - is derivable from S.

Given D, the second guesser can reason, in a "given what I know now" manner, about two related issues:

Determine which element of D to reconsider.

Reconsider, find a different selection for the chosen decision item.

To elaborate, suppose the program has suddenly realized that it has over allocated floor space by 20,000 sq. ft. The set S turns out to be $\left\{S_{1C} \cdots S_{4C}\right\}$. The set D is then $\left\{D_1 \cdots D_4\right\}$. Examination of D reveals that D_3 is the best decision level item to reconsider for several reasons: S_{3C} has many elements that are more prudent in economizing space, and D_3 is specific (as in the abstraction hierarchy shown in figure 1). S_3 could then be scrutinized using the added information that an acceptable choice would have to consume at least 20,000 sq. ft. less than S_{3C}. Note that in second guessing here PLANET has access to information that was not available when it first attempted to process the decision level item D_3. Quite literally, it is using hindsight.

The set of choices in S are indeed responsible for the over constrained state; however, we claim that the identification of a scapegoat in S and its replacement is made more rational by examining each selection level candidate in the context of its decision level choice point. Furthermore, each of the decision level elements should be compared and contrasted. This distinction is important -- the selection level items are the retractable "assumptions", but second guessing should pivot around the decision level items. We will now review why we have found it elusive to make this distinction explicit in two of the better known data dependency utilities.

3.2. A Critique

The Truth Maintenance System[6] of Doyle (1978, 1980) has the capability to deny some *non-monotonic* belief that entails a *contradiction*. The result is to make the contradiction go away. The selection of the

[6] Doyle has noted that this name is not really right and has opted the use of Belief Maintenance or Reason Maintenance, but the term Truth Maintenance has caught on in the community so we hereby acknowledge the misnomer and continue to use it.

culprit and the construction of its denial is arbitrary; or as noted in Doyle (1980) the backtracker is engaging in a blind search while trying to resolve the contradiction. There do exist clever methods for imposing some structure on the order in which alternatives are examined (see section 8 of Doyle, 1978). To make those methods apply in our context, the decision level items must be treated as non-monotonic beliefs, that is, assumptions in the TMS sense -- otherwise they would not be considered as choice points. However, as we have pointed out, the decision items are not really assumptions that the problems solver has made, but are forced choices that the problem solver cannot retract. Rather, it is the selection level items that are subject to retraction. As we will see, it is not sufficient to simply give the selection level items non-monotonic support.

Now if we give the decision level items monotonic support and only the selection level items non-monotonic support, the information about tradeoffs that is contained at the decision level is lost. This problem would be partially resolved if a user-supplied procedure could be called to select the culprit. We explored this possibility only briefly when designing PLANET because it leads to another set of problems -- the backtracker is called during the midst of the Truth Maintenance procedure; thus, the user supplied code would be executed when the belief set is in an unstable state.

The Reason Utility Package, RUP (McAllester, 1982), notes contradictions whenever it is no longer possible to satisfy some logical relationship. RUP does not support nonmonotonic reasoning, rather it allows for the retraction of some premise where a premise is an asserted, as opposed to a derived, proposition[7]. By default RUP will choose the premise to retract by querying the user, however the right kind of hooks do exist to allow an application specific procedure to make the decision.

We were initially attracted to the possibility of designating a domain specific procedure for deciding how to handle an over constrained resource plan. It soon became apparent that what we really wanted to do was still elusive. To be able to make the selection level choices retractable they needed to be asserted in RUP as premises. Unless we created explicit attachments[8] for dependency information between decision and selection items, we cannot retain (in terms of dependency) the context imposed on a constellation of selection items by their corresponding decision choice point[9]. Maintaining such attachments would mean duplicating the truth maintenance mechanisms already in the system.

Even if the inelegance of this approach were tolerated, the interface machinery required to keep the two dependency systems in sync would be

[7] RUP itself is a constraint propagation system. Given a logical relationship, a known truth value of some term will constrain the truth value of other related terms. When a relationship becomes over constrained, a contradiction is signaled.

[8] RUP does provide for user-defined attachments to individual terms.

expensive in time and space.[9]

To summarize, our objective is to represent the dependency of a selection level choice in the context provided by the decision level problem. When a resource plan becomes over constrained, we want to view selection level maneuvers within their decision level context, as well as other decision level choices that influence the over constrained state. In trying to achieve this using available systems, we found an asymmetry of problems. In Doyle's TMS, we found considerable flexibility in building justification structures to build dependencies but the backtracking procedure was weak. In McAllester's RUP, we found considerable flexibility in the backtracking procedure but the methods for representing the dependencies were cumbersome.

4. Toward Reasoned Assertion and Retraction of Assumptions

As the preceding discussion suggests, the problem solver needs structures that will explicitly maintain enough context about choice points -- specifically, information about tradeoffs among alternatives -- whenever forced choices are made. Then, if unacceptable situations arise, the program can use this plus information on the type of constraint violation to determine which assumptions to retract to best alleviate the problem.

[9] i.e. for each dependency in the system a few bookkeeping "nodes" would be created for each relation, and demons to communicate a state change in one system to the other would be needed.

Each of the choices shown in Figure 2 is represented as a structured object (enclosed within braces):

```
{
  node-id : I state1_i
  decision : D_i
  selection : S_i
  alternatives : {S_1,...,S_n}
  disadvantages : ((S_1(X1X2..Xn))(S_2(X1X2 ··· Xn)))
  advantages : ((S_1(X1X2..Xn))(S_2(X1X2 ··· Xn)))
  o
  o
}[10]
```

The "advantages" and "disadvantages" slots are both lists of dotted pairs; the first element in a pair is a selection item (an alternative to the choice), and the second is a list of resource categories for which the alternative is advantageous (or disadvantageous).

Whenever an unacceptable allocation of resource, X, arises, the program invokes a two step procedure to determine its revised set of choices. First, all decision level items disregarding X as a disadvantage are recorded. Then, the combined pool of selection level items are compared to determine the selection best alleviating the underlying problem.[11]

Recall the example given earlier and shown in Table 1. In this

[10] There are also "slots" containing data dependency information not shown here.

[11] In situations where dependencies exist, resource impacts of undoing the dependent choices are also taken into consideration when evaluating the various maneuvers.

situation the program has configured the model with the prerogative of preferring space savings over the others. It then discovers that all capital resources have been exhausted. There are three decision items that have contributed to the over-constraint (table 1-a through 1-c).

By looking at these tables we can see that S-Test (Table 1-b) is the best apparent candidate to reconsider because the biggest potential savings can be realized by reconsidering a new choice in S-Test. That is, we can save at least $3 million by considering S-Test as opposed to saving at least $2.5 million by considering MIF-test or at least $2 million by considering Insertions.

PLANET will identify these decision choices as the ones to consider and will pick S-Test as the particular decision to reconsider.

The structured object representation of the S-Test decision (with the internal identifier as state-15) would be:

```
{
   node-id : state-15
   decision : (modules: S-Test)
   selection : L-20
   alternatives : (QV FC-33)
   disadvantages : ((QV labor capital)(FC-33 labor capital))
   advantages : ((QV space)(FC-33 space))
   o
   o
}
```

The germane portion of PLANET's "state space" when the over constrained situation is noted is shown in figure 2a. Via detailed examination of dependency information, PLANET correctly identified these three decision points as having disregarded capital. The important aspect of this examination is that it had a specific goal: Identify the decision points that entail the over constrained situation that also disregard the down side on capital. In effect, the procedure focuses on a problem specific set of contributing factors instead of merely finding some basic set of entailment.

Given the three possible decision points to consider, PLANET then examines the available set of selection choices to determine which one to reconsider. To do this the program applies hindsight using the current state of the model and domain specific knowledge of resources and resource tradeoffs. It is effectively doing a "given what I know now" analysis of the alternatives in each decision object[12] and picking an item to reconsider.

The structure of the "state space" after picking S-Test as the decision item to reconsider and retracting the S-Test selection appears in figure 2b. In effect, the choice revision process involves establishing a "macro move" (Figure 2b) that bridges the state on the left of the retracted choice to the set of good choices generated chronologically after it. Having alleviated the problem, choices following the set of good choices are made

[12] Sending them messages to find out about their current estimates of consumption.

in light of the updated resource situation.

PLANET routinely makes these kinds of revision. When faced with the task of reevaluating previous choices, it applies both problem specific and domain specific knowledge to identify the choice. In doing so, it uses hindsight - the current problem state - to identify its backtracking point. Like other dependency directed backtrackers (Stallman and Sussman, 1977; Doyle, 1980; McAllester, 1982) only those inferences dependent on the choice are retracted.

PLANET is also capable of using other kinds of knowledge than we have demonstrated here. For example, if several equally good decision level items are identified it will examine them in light of other resources that are nearing exhaustion. Thus, it is capable of recognizing and avoiding other potential problems while it is second guessing.

5. Limitations and Concluding Remarks

In the example considered above, a detailed comparison of the resource implications of alternative selections was possible. Unfortunately, this is not always the case. If the set of potential backtracking points identified by the program after its first step in the procedure include choices about the more abstract parts of a model where detailed resource requirements have not yet been assessed, a quantitative comparison is not possible. Projecting the consequences of the various maneuvers is then difficult. Further, if long chains of dependencies exist, the program might pick as its best move one that involves undoing large parts of the partial model since this would free up the maximum amount of the scarce resource.

Deciding how much of the model to undo is complicated when a good action will alleviate the problem marginally, allowing it to recur a few steps later.

Finally, a limitation of the existing backtracking scheme is that the program is unable to recognize situations where it might be better off retracting a *combination* of choices as opposed to a single decision.

We are currently working on ways by which domain specific knowledge pertaining to these criteria may be represented in terms of dependency information and made accessible to the backtracker.

In conclusion, the issues raised here have been driven by a complex, real-world problem where existing formalisms proved to be useful but inadequate in modeling the essential nature of the problem. We have developed a scheme whereby a backtracker might assess more rationally the reasons for an untenable situation, and modify its existing set of choices in light of the evolving scenario of constraints. While the methods outlined above are preliminary, they represent a step toward a more general method for reasoned introduction and retraction of assumptions for decision situations where tradeoffs are involved.

References

de Kleer, J., Doyle, J., Steele, G. and Sussman, G., AMORD : Explicit Control of Reasoning, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, 1977.

Dhar, Vasant., PLANET: An Intelligent Decision Support System for the Formulation and Investigation of Formal Planning Models, Ph.D. Thesis, University of Pittsburgh, 1984.

Doyle, Jon., Truth Maintenance Systems for Problem Solving, TR-419, Massachusettes Institiute of Technology, Artificial Intelligence Laboratory, 1978.

Doyle, Jon., A Truth Maintenance System, *Artificial Intelligence,* June, 1979.

Doyle, Jon., A Model For Deliberation, Action, and Introspection, MIT-AI TR-581, May 1980.

McAllester, D., Reasoning Utility Package, AI Laboratory Memo 667, April 1982.

McDermott, Drew. and Doyle, Jon., Non-Monotonic Logic I, *Artificial Intelligence* Nos 1 and 2, April 1980 (special issue).

Quayle, Casey., Object Oriented Programming in Franz Lisp, Working Document, Decision Systems Laboratory, University of Pittsburgh.

Simon, Herbert., On Reasoning About Actions, *Artificial Intelligence,* June, 1979.

Stallman, Richard. and Sussman, Gerald., Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, *Artificial Intelligence,* volume 9, No.2, October 1977, pp 135-196.

Stefik, Mark., Planning with Constraints, Stanford University, Computer Science Department, STAN-CS-80-784, 1980

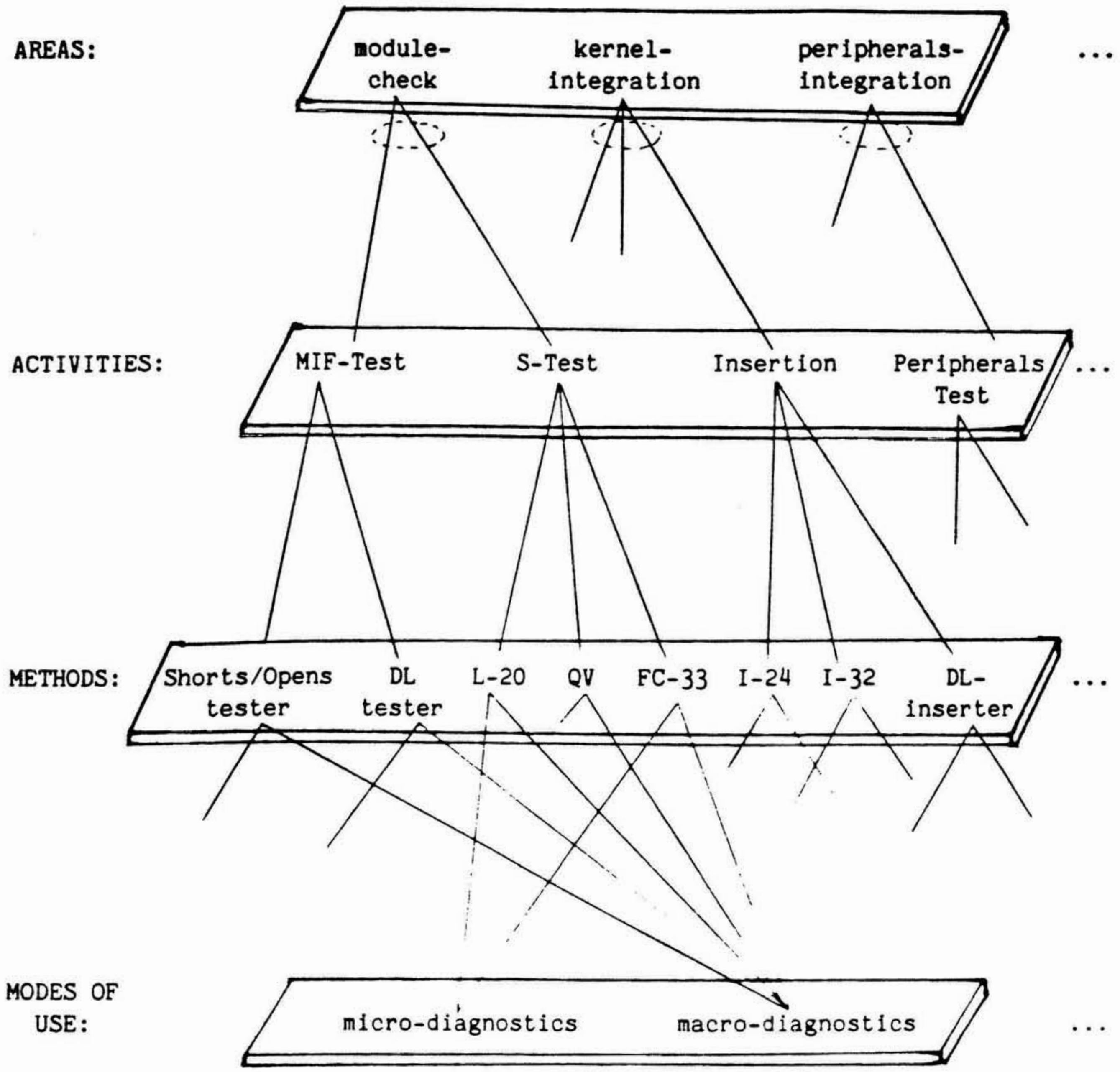

Figure 1

A schematic of the hierarchy of choices in some areas of the CMC computer manufacturing process. For readability, we have not shown all choices involved in the various areas, activities, methods, etc.

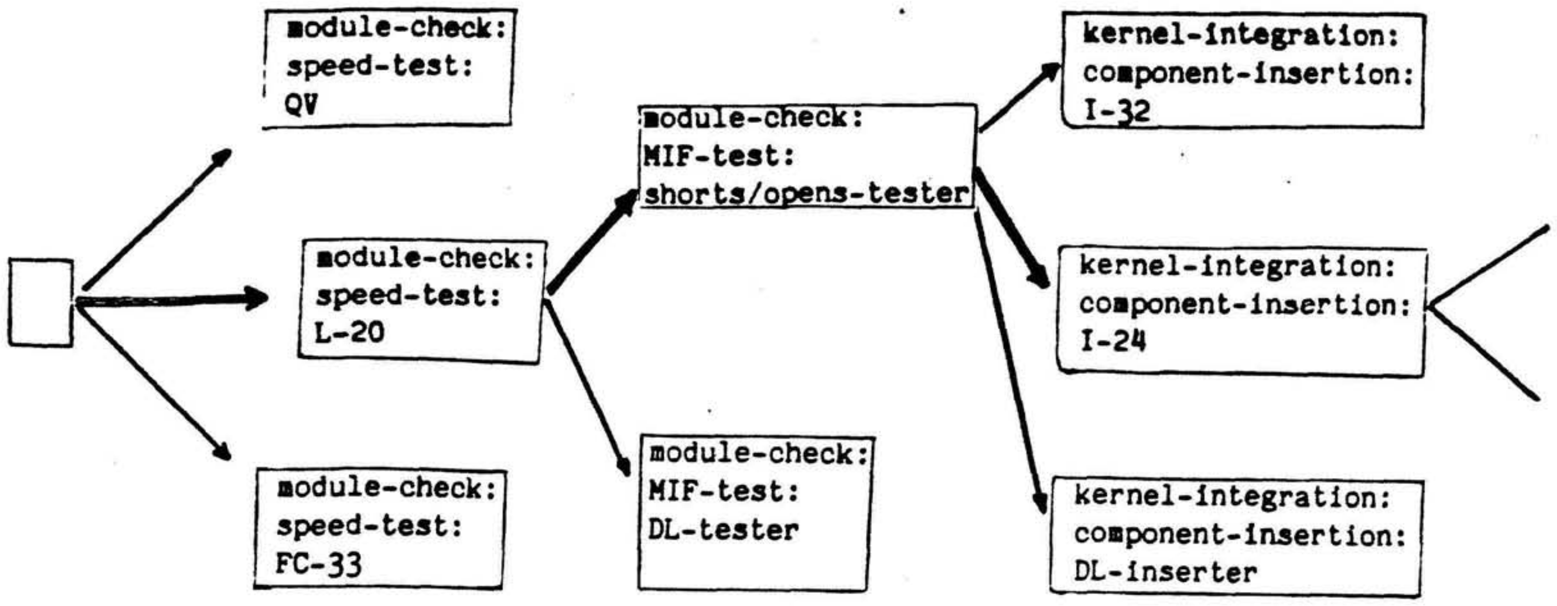

Figure 2a

The dark line indicates those nodes that have been expanded in the state-space using a heuristic evaluation function. Regular lines connect states that have not been explored further.

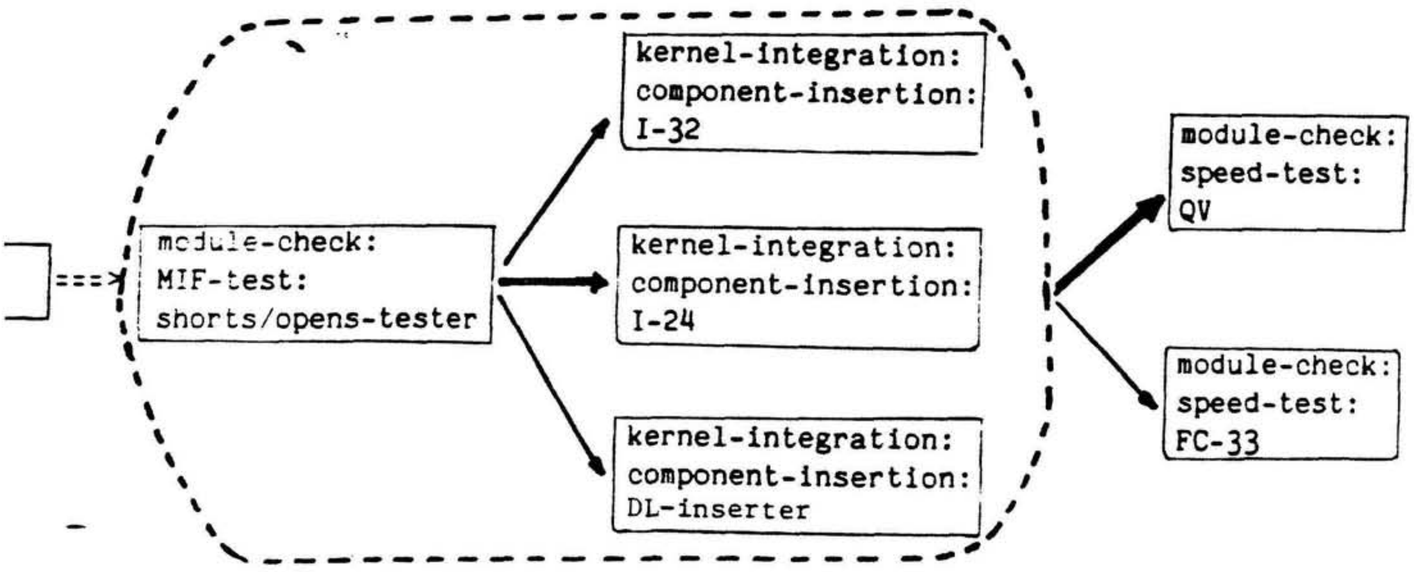

Figure 2b

The dashed ellipse encompasses those choices (made chronologically after the "bad" choice) that will not be retracted. The "===>" is a "macro-link" that connects the state on the left of the bad choice to set of "good" choices generated after it which are preserved. The state-space search continues to the right of the ellipse -- in this case, the program must make a choice from its reduced set of alternatives for module-check:speed-test.