Choices Made by a Planner: Identifying Them, and Improving the Way in Which They are Made

by D. Croft.

Errata Sheet

24th June 1985

Page 30: dia. 2. 9 - the link to the "Put A on B" node should actually be attached to the preceding join (J) node.

Page 33: dia. 2. 11 - the Y(es) and N(o) paths from the choice "Any expandable nodes introduced by these expansions" should be switched, so that if there are no remaining expandable nodes, the system will go on to satisfy the unsupervised conditions.

Page 177: dia. 6. 2 - node 5 should be "goal on(b, c) = true".

Choices Made by a Planner: Identifying Them, and Improving the Way in Which They are Made

D. Croft.

M. Phil University of Edinburgh 1984

Abstract

This thesis discusses the ways in which choices are made by an AI planner. A detailed examination is made of the prerequisites for choice making, and a discussion of how the making of good choices can be automated is included. For a given planner, the prerequisites for choice making can be split into two parts: finding the types of choice made during the planning process, and finding the information most relevant to the making of each type of choice. Two means of automatically making "good" choices are described: using general planning policies that have been supplied by the user, and using learned heuristics. These possibilities are explored for a non-hierarchical version of Tate's NONLIN.

Acknowledgements

This work was carried out under a studentship from the Science and Engineering Research Council. I am grateful to the Machine Intelligence Research Unit and Intelligent Terminals Ltd. for office and computing facilities. The Deparment of Artificial Intelligence found me a supervisor, Dr. Austin Tate, who prevented me from biting off more than I could chew, and provided invaluable guidance at the critical points in my work. I must also thank Professor Donald Michie who, perhaps unwittingly, led me to the field of planning. Lastly, it would be a poor show not to mention Alen Shapiro and Colin Lindsay, whose digital fingers kept our computer going through thick and thin.

Table of Contents

Table of Figures

CHAPTER 1

Introduction

1.1. Overview

Many planners do not explicitly acknowledge that choices are made during the planning process. Choices are recorded and ordered by some secondary search process, which may not be under intelligent control. For instance, there are usually several different ways of refining an action or correcting an interaction between conflicting goals, and the planner must choose the best. When and how such choices should be made is generally "hard coded". This is undoubtedly a disadvantage if the user wishes to change the control structure of the planner. However, more important from the point of view of the current discussion, the explicit and uniform representation of the choices made during planning opens up the possibility of guiding the planner, so that the *best* decisions are selected. In this thesis, three methods of guiding the making of choices are considered:

(1) Interactively, allowing the user to do the thinking;

(2) Under the control of user-defined general planning policies;

(3) Under the control of heuristics that have been learned from examples.

The guide will need to be well informed, whether it is human or machine. But it must not be overwhelmed with unnecessary information. Hence, information from the plan and elsewhere must be presented in an appropriate form when a choice is to be made. A program, called MODPLAN (MODular PLANner), has been written in PROLOG to make explicit the choices made during planning.

1.2. MODPLAN: Primary Aims

The main aims in writing MODPLAN were:

(1) To separate the choice making that goes on in planning from the actual details of the planning process.

(2) To enable the user to keep a record of abandoned paths in the planner's solution space, so that it is possible, at some future date, to return and recommence planning at the point at which it had been left off.

(3) To enable intelligent guidance of the planner at choice points

(4) To make the planner as modular as possible, so that it is easy to incorporate new choice types, heuristics etc.

(5) To include in the program listings a detailed set of comments, so that the program could be used as the basis of future work by other people.

1.3. Thesis Outline

The work presented here was influenced in many ways by the work of other people in the fields of problem solving and learning. A review of some of this work, with an indication of how it relates to MODPLAN, is given in chapter 2.

A primary aim of this thesis was to make explicit the ***types*** of choice made by a planning program, and to attempt to provide facilities for automatically guiding choice making. The first of these is a necessary prerequisite for the second. Thus, a program was written whose operation was centered around the choices made during plan generation. This is described in chapter 3. This chapter also contains a description of the Plan Modification Formalism written for MODPLAN. This is used to implement the effects of choices once they have been made, and gives the experienced user a simple language for defining operators that can change plans.

As a by-product of making choices explicit, a means had to be developed to make the control structure of the planner partially independent of the backtracking mechanism of PROLOG. This used a tree structure to store all of the choices made, and opened up the possibility of permitting the user to restart planning in any part of the search space that had already been explored. An interactive system for plan restarting (or "refocusing") was provided. It allowed the user to locate a target choice-point by specifying the features that it should have, and is documented in chapter 4.

Chapter 5 describes the automatic choice making mechanisms for MODPLAN. If choices are being made automatically, then there are at least two classes of choice making algorithm. First, there are algorithms that form the specific application of a general planning policy (also known as ***operationalisations*** - a term due to Mostow ([MOSTOW, 1979])). Second, heuristics that had been learned from examples of choices made previously could be used. The example choices would be selected, by the user, from the tree of choice-points built up by MODPLAN during planning. The techniques developed for extracting the information needed in automatic choice making, and a simple language for representing the choice making algorithms (the "Operationalisation Formalism") are described in this chapter.

To illustrate how changing the way in which choices are made can affect planning, a number of experiments were performed, and are documented in chapter 6. Various choice making algorithms, representing operationalisations of a number of general policies, are used to make choices for certain choice types. The chapter shows that a user can use data on the performance of the planner under previous suites of policy operationalisations to *improve* the planner's performance, through incremental changes to the current suite of policy operationalisations. Investigating the way a human makes these incremental improvements would be a first step in automating such a process.

A chapter considering MODPLAN's generality is included. In it, the problems of "modelling" planning systems other than NONLIN are examined. An outline of the changes that would have to be made to MODPLAN to accommodate three other systems, WARPLAN, MOLGEN and dependency directed backtracking, is presented.

Two glossaries are given as appendices. The first contains various general planning terms, plus many terms special to MODPLAN. The second gives a list of PROLOG terms referred to in this thesis that may not be familiar to the reader. The third appendix explains the way in which NONLIN's Task Formalism and plan representation have been rendered in MODPLAN. A fourth appendix, containing examples of how the various formalisms described in this thesis have been transcribed into PROLOG, is also given.

1.4. Relationship of NONLIN to MODPLAN

It was felt that, rather than writing a planning program from scratch on which to base the work on guidance at choice points, it would save time to model an already existing program. Of the many planners documented in the AI literature, it was decided that NONLIN would be a suitable choice. NONLIN has recently been used as the basis of the DEVISER system, which plans missions for deep space probes ([VERE, 1981]). NONLIN has many state of the art planning features, and there is fairly good documentation available ([TATE, 1976, DANIEL, 1982]).

1.5. Why Use PROLOG?

MODPLAN is written in PROLOG ([CLOCKSIN, 1981]), a logic programming language. A central feature of many planners is the use of pattern matching and database search for selecting appropriate operators. PROLOG is a good language for creating databases, since it allows data to be represented in the same way as the program; furthermore, pattern matching is the way that PROLOG itself selects the functions which it uses, hence no special database package need be written for the planner. PROLOG also has a convenient list representation, useful for storing large assemblages of information, such as the plan network. Programs written in PROLOG are easily extensible, and PROLOG has reasonable debugging facilities.

CHAPTER 2

Previous Work

2.1. Preamble

As a preamble, it would be of interest to look at some other work relevant to this thesis. This could broadly be categorised into two parts; planning systems, and learning programs.

A number of early planning programs, such as Graph Traverser, STRIPS and WARPLAN, are included at the beginning of the review, to put the later programs in some kind of historical perspective. We shall look at a number of non-linear planners, which generate plans that are partially ordered networks of actions, because the planner modelled by MODPLAN (Tate's NONLIN) works in this way. The work of McDermott and Wilensky on the representation of policies influenced the choice making mechanisms used by MODPLAN. This work is thus examined in some detail.

The section on learning programs is somewhat shorter than the planning section, since no actual learning of heuristics has yet been incorporated into MODPLAN. However, a number of approaches to the learning of rules by example are discussed, and their suitability as methods for learning choice making heuristics is considered. Also in this section is an examination of the work of Mostow in the field of advice taking. His concept of operationalisation, already mentioned in the introduction to this thesis, is used as a way of making choices under the control of general policies.

2.2. Review of Selected Planning Programs

2.2.1. Introduction

2.2.1.1. General

Planning is a sub-field of the more general subject of problem solving. The usual approach to problem solving is to assume that the problem solver has, for a given domain:

(1) A description of the initial state of the world in which the problem is to be solved,

(2) A description of the operators which can be applied to the world to change it's state, and

(3) A description of the desired final state of the world (or a set of facts to be removed from and added to the initial state - the goals of a problem solver).

The solution to the problem will be an ordered (or partially ordered) set of operators (called a ***plan***), which transform, by stages, the initial state into the final state of the world. The stages in the planning process are called ***steps***.

The way in which planning systems have been designed has depended on their designer's particular interests, but the following sections give a number of broad categorisations.

2.2.1.2. Use of Domain Specific Information

The amount of domain specific knowledge used, and the way it is used is one way of categorising planners. Early planning systems, such as GPS and STRIPS made an attempt to guide the planning process using completely general heuristics, so that domain specific knowledge only appeared in the operator definitions. State space search programs, such as Graph Traverser ([DORAN, 1966]), did use heuristics to guide their search, but such heuristics yielded only a numeric estimate of the best path to follow. Later systems, such as PLANNER and CONNIVER embodied more domain specific knowledge, and used pattern directed procedure invocation techniques. Hearsay II (and it's derivatives), and others, employed domain specific heuristics, known variously as critics, knowledge sources etc., to guide the search for a solution.

2.2.1.3. Linear or Non-Linear?

The way in which plans are built up can also be used to categorise the planner. Two basic methods are distinguishable: linear and non-linear. Linear planners attempt, by some method, to build up a continuous sequence of actions, which, if executed, would transform the initial to the desired world state. However actions might be inserted into this sequence, the assumption is always that there must be a strict ordering on these actions. Non-linear planning systems make no such assumptions. In fact, the initial set of goals is assumed to be solvable in parallel, and it only as the planning process proceeds that orderings are imposed on goals and actions.

2.2.1.4. Single Level vs. Hierarchical Planning

The way that planners handle details varies too. There are two approaches, hierarchical and non-hierarchical. Non-hierarchical planners assume all information to be equally important. Hierarchical planners assume some pieces of information to be more important than others; generally, they will be given, or will generate, a ranking of facts on order of importance. When planning occurs, the initial plan will only use the information that is most important - a high level, undetailed plan will be produced. For each step in this plan, more detailed plans will be made, using information ranked as less important than that used in the higher level plan. This process will be repeated for each level in the ranking. Thus, a hierarchy is produced, in which each level corresponds to a more detailed representation of the plan. The advantage of doing it this way is that the search for a solution at any particular level is not cluttered with unnecessary detail, and can thus proceed more quickly than a non-hierarchical planner would be capable of (a case of being able to see the wood in spite of the trees). However, by leaving out detail, we are making an approximation, and hence opening up the possibility of errors at the higher levels of the planning process. So which information is treated as detail, and which is regarded as important must be carefully thought about beforehand. ABSTRIPS (Sacerdoti, [SACERDOTI, 1973]) was an early attempt to put such ideas into practice, using the STRIPS planner as a basis for a hierarchical system. Most state of the art planners use some form of hierarchical structuring.

2.2.1.5. The Examples

To make these ideas more concrete, we shall look at some examples.

Five examples of general, linear planning systems will be given here. The first, Graph Traverser (Doran and Michie, [DORAN, 1966]), illustrates the state space search approach to planning. The problem to be solved is represented as a graph, in which the nodes correspond to the world states, and the arcs to operators. In the planning domain, such graphs would be directed, since operators can only work in one direction. A detailed discussion of such problem solving methods is given in Nilsson [NILSSON, 1971]. The other examples are intended to illustrate an approach known as subgoaling. Planners which work by subgoaling attempt to find intermediate sets of goals, which are easier to achieve than the desired set of goals, and to plan for these. Having achieved the intermediate goals, they then try to generate a plan to achieve the final goals from the intermediate state. This can be a recursive process, since the subgoaling approach can be applied to the subgoals, etc. The programs to be discussed here will be:

(1) STRIPS (Fikes and Nilsson, [FIKES, 1971]).

(2) WARPLAN ([WARREN, 1974]).

(3) INTERPLAN ([TATE, 1974]).

(4) Waldinger's goal regression method.

PLANNER (Hewitt, [HEWITT, 1970]) is a planning system falling into the group of planners that use knowledge in a procedural form. Pattern matching is used extensively in this program, a facility which all present day planners now employ.

NOAH and NONLIN are both non-linear planners. NONLIN is discussed in some detail, since it is modelled by MODPLAN.

Daniel ([DANIEL, 1977]) and Hayes ([HAYES, 1973]) both worked on a structure called the decision graph, in which decisions made by the planning program were stored. This is very similar in many ways to the choice point tree used by MODPLAN, so a brief discussion of the above work is included.

An unusual problem solving program, called NASL, was written by McDermott in the mid-seventies. This program only produced small plans at an abstract level; as soon as an action became specialised enough to be executable, it was executed. Thus, no space consuming trail of the program's choices had to be kept. Another interesting feature of this program was it's use of secondary goals, also known as policies.

The separating of choices from the planning program, and the use of policies to guide choice making, owes much to the meta-planning concepts of Stefik and Wilensky. Short reviews of MOLGEN and PANDORA have been included, to give an outline of these ideas.

One of the features that was felt to be important for MODPLAN was the ability to be highly interactive. Few planners seem to put much importance on this. However, if a planning program is going to be of any practical value, this is essential. Some of the facilities of Wilkins' SIPE ([WILKINS, 1981, WILKINS, 1983]) have been incorporated and extended in MODPLAN. This program is also reviewed below.

2.2.2. Graph Traverser

Graph Traverser is potentially applicable to any domain in which operations can be represented as a graph search process. It works as follows. The user first supplies it with two functions, **develop** and **evaluate**. **develop** is the function which defines how the graph is to be grown. In a problem solving domain, this would define which world states can be reached from the current world state. **evaluate** looks at a world state, and evaluates it's distance from the goal state (the "cost" of attaining the goal state from the current state). It is a heuristic, presumably selected after much experimentation, which should direct the Graph Traverser to the correct solution to the problem as quickly as possible.

Given these functions, Graph Traverser goes through the following cycle:

(1) Set the node currently in focus to the first node in the search space.

(2) If the node currently in focus has a zero cost, a solution has been found, so stop.

(3) **Develop** a set of child nodes, and **evaluate** the cost of getting from each child to the goal state.

(4) Select the "cheapest" child node. If it's cost is zero, then it is a solution, so stop. Else, **evaluate** the costs of getting from each of the grandchildren to the goal.

(5) If any of these grandchildren are cheaper than the node currently in focus, change the node currently in focus to that grandchild and go to 3, else choose the next cheapest of the children of the node currently in focus, and go to 5.

Graph Traverser will thus grow branches in the state space of the problem, until it comes across the solution world state (for which the cost of attainment is zero). The efficiency of this system is critically dependent on the **evaluate** heuristic. A great deal of work has been done in constructing suitable classes of evaluation function. See Nilsson [NILSSON, 1971] for a discussion of this.

2.2.3. STRIPS

The second general planner to be discussed is STRIPS. World states are described in terms of predicate calculus well formed formulas (wff's), which represent the information as facts and rules. STRIPS operators are described by the user in schemata. An example of such a schema is given in dia.2.1

This defines an operator in a simple robot world, in which a wheeled robot can push boxes around. This particular operator will cause a box k to be pushed from location m to location n. The definition splits into two different parts: the condition part, and the effects part. The condition part specifies the preconditions which must hold before this operator can be applied - here, the robot must be at location m (ATR(m)), and the object k must be at location m also (AT(k, m)). The effects specify what changes the operator makes to the world. These are given in terms of a delete list and an add list. The delete list tells us which facts must be removed from the world model, and the add list tells us which facts must be added.

When looking at a problem, STRIPS will first compare the initial world state with the goal state. From the comparison, a set of differences (similar

```
push(k, m, n)
        Precondition:  ATR(m)
                       & AT(k, m)
        delete list    ATR(m);
                       AT(k, m)
        add list       ATR(n);
                       AT(k, n).
```

dia.2.1 Box pushing operator definition

to those generated by an earlier system, GPS [NEWELL, 1975]) is extracted, in the form of a partial proof, describing the way initial and goal states differ. These differences are used to select an operator which will *reduce* the differences. Achieving the preconditions of this operator become the new subgoals of STRIPS. STRIPS now repeats the process just described recursively, until a plan achieving these preconditions has been generated. STRIPS will then try to find a set of differences between the world state achieved by the first subgoal operator and the goal state. This will lead to the selection of a new operator to reduce these differences. As before, the preconditions of this new operator form subgoals, and the planner will now try to satisfy these. This process continues until a complete plan is generated.

An example of how this process can be represented is given below.

(1) The initial state of the world model is S_i. The desired goals to be achieved are G_0. This situation is shown in the following diagram.

$$S_i \longrightarrow G_0$$

(2) The first thing STRIPS does is to find the set of differences between S_i and G_0, using this to select the operator Op_1 to reduce them. The preconditions of this operator, G_1, are taken as the new subgoals of STRIPS, giving rise to the following:

$$S_i \longrightarrow G_1 \xrightarrow{Op_1} S_1 \longrightarrow G_0$$

(3) STRIPS now uses the difference between S_i and G_1 to select operator Op_2. It discovers that the preconditions of Op_2, G_2 are true in S_i, so the world state resulting from the application of Op_2 can now be computed,

S_2. This is shown in the diagram below.

$$S_i \longrightarrow G_2 \xrightarrow{Op_2} S_2 \longrightarrow G_1 \xrightarrow{Op_1} S_1 \longrightarrow G_0$$

(4) G_1 are true in S_2, so now the differences between the first subgoal and the desired goal are found, and the operator Op_3 selected to reduce them. The situation is now:

$$S_i \longrightarrow G_2 \xrightarrow{Op_2} S_2 \longrightarrow G_1 \xrightarrow{Op_1} S_1 \longrightarrow G_3 \xrightarrow{Op_3} S_3 \longrightarrow G_0$$

The preconditions of this operator, G_3, are true in S_1, so the world state resulting from the application of Op_3 can be found ($\cancel{G_3}$ S_3). The desired goal state, G_0, is true in this world state, and hence the problem is solved, the plan being the sequence of operators, Op_2, Op_1, Op_3.

2.2.4. Warren's WARPLAN

WARPLAN was probably the first attempt to realise a planning program in PROLOG. It was written by David Warren in 1974 ([WARREN, 1974]) in an early version of PROLOG.

The basic strategy of WARPLAN is to take a conjunction of goals ("goals" and "facts" are used almost interchangeabley here, as are "operations" and "actions") and to attempt to solve each one separately. WARPLAN has three main data structures:

(1) The database, a set of assertions defining:

a) the add sets, delete sets and preconditions for the operators,

b) conjunctions of facts which are always true and conjunctions of facts which are impossible, and

c) the initial state of the world.

(2) A time ordered sequence of operations (actions).

(3) A conjunction of protected facts, representing goals already achieved within a given invocation of the planning function.

When WARPLAN tries to solve for a goal, it will first check to see if the goal is a fact which is always true, or whether it holds in the current state of the world. A goal holds in the current state of the world if either (i) it has a database entry in the current state, or (ii) the sequence of actions so far generated transforms the initial world state into a state containing that goal. Since (i) will probably not be the case, unless the current state is the initial one, the holds test will check each action on the list of actions achieved, to see if the goal is on it's add set, making sure that it is not deleted by that action. If an action is found which deletes this goal, or it is not in the add set of any of the already achieved actions and it did not hold in the initial state of the world, then WARPLAN will look for a new action in whose add set this goal is a fact. It will then attempt to achieve this action. There are two ways in which it may do this:

(1) If the action does not delete any of the already achieved (protected) facts, then the preconditions for the action are determined, and are set up as a set of goals for a new invocation of the planning function to solve. This action is then added to the end of the existing action sequence.

(2) If the above is not possible, it may be feasible to insert the action somewhere else in the current plan. The procedure is recursive:

a) Strip the last action from the plan, making sure that this action does not delete the goal to be achieved (everything fails if it does!).

b) Retrace the set of protected facts. This procedure removes any members of the last action's addset from the original set of protected facts, and adds on the preconditions of the last action to this new set of facts. The resulting set of facts, if protected, will firstly preserve any goals already achieved at the last level of recursion of the planning function, and secondly, prevent the deletion of the facts which make up the preconditions of the last action.

c) Now the achieve procedure is called again (this is the recursive bit), to see if the action can be inserted at this point, or whether more actions will have to be removed from the plan, to insert the action earlier on in the sequence.

Having thus solved for the first goal in the sequence, WARPLAN then strips off the second goal in the conjunction and attempts to incorporate any actions required to achieve this into the plan generated for the first goal. This is repeated for each goal, the partial plan and the set of achieved (protected) facts building up all the time. When it has planned for the last goal, WARPLAN stops, and prints out the plan which it has generated.

A note on some of the functions used by WARPLAN. ***plan*** is the core function, which generates a plan given a conjunction of goals. ***solve*** incorporates into the current partial plan any action or actions required to achieve a given goal. ***achieve*** achieves the action specified by ***solve*** for realising the goal. ***holds*** checks if a fact holds in the current state of the world. ***retrace*** generates the set of protected facts which must hold *before* the last action in a plan, given the action and the protected set holding *after* that action.

2.2.5. INTERPLAN

2.2.5.1. Introduction

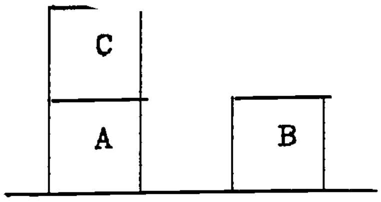
Austin Tate developed a linear planner, called INTERPLAN ([TATE, 1974]), to deal with problems in which interactions arose between goals during problem solving. To go with this planner, he developed a "holding period" representation for goals, which could be used to specify notional periods of time over which goals should be true. By reordering goals or extending their holding periods ("promoting" them), problems caused by interactions could be eliminated.

2.2.5.2. An Example

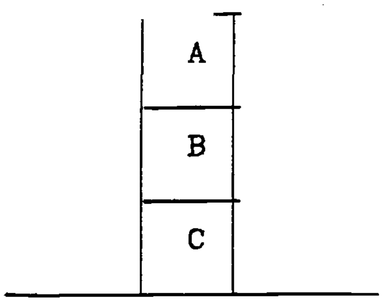
As an example of how the promoting of goals can eliminate interactions, take a blocks world problem in which a block C is initially on top of a block A, and block B is initially standing by itself (see dia.2.2). An "on top of" relationship will be described in the following example by on(X, Y), which indicates that block X is on top of block Y. A "clear top" relationship will also be used. cl(X) means that the top of block X is clear, so that other blocks may be put onto it.

Our problem is to achieve the conjunction of goals on(A, B)&on(B, C) (see dia.2.3). If we attempt to achieve these goals in isolation, in either order, we end up with plans containing redundant actions. The reason for this is that the solutions for the two goals *interact*.

A possible initial holding period diagram for the problem is shown in dia.2.4. The arrows indicate the periods over which the goals on the left hand side must hold. The action sequence is comprised of the actions, in chrono-



dia.2.2 Initial state for blocks world problem



dia.2.3 Desired final state for blocks world problem

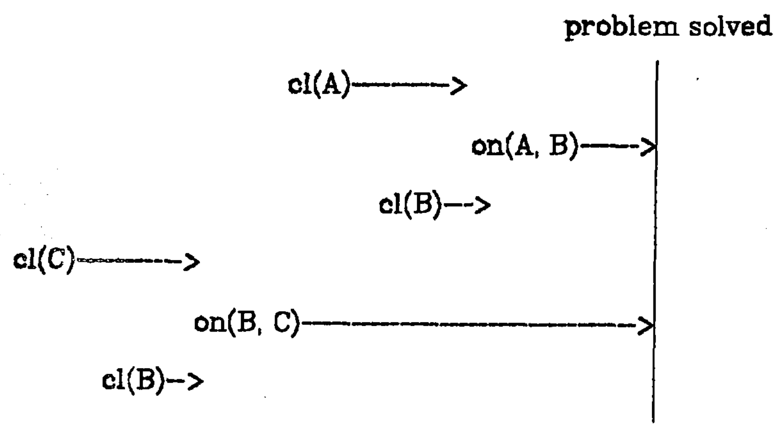
logical order, that would achieve these goals.

```
                              problem solved
                    on(A, B)---------------->|
          on(B, C)-------------------------->|
Action Sequence: PUTON(B, C), PUTON(A, B)
```

dia.2.4 Holding periods for top-level goals

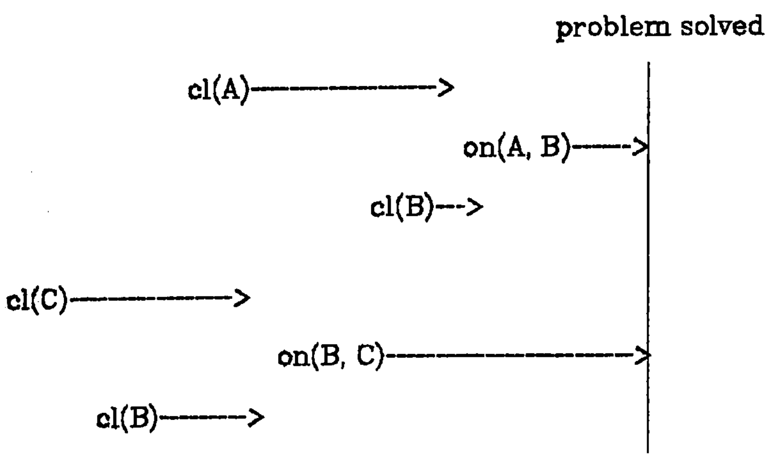
The two operators, PUTON(B, C) and PUTON(A, B) have been chosen by the system to achieve these goals, so we can now put the preconditions of these operators into the diagram as sub-goals (dia.2.5).

B and C are clear in the initial world state, so we can achieve on(B, C) straight away. Unfortunately, this makes cl(A) false over a period during which we need cl(A) to be true, if we are to successfully achieve on(A, B). An *interaction* has occurred. An unintelligent planner, such as STRIPS, would try to circumvent this problem by inserting more operators into the plan. INTERPLAN uses a different strategy; it *promotes* one of the interacting goals, as shown in dia.2.6. The system must now achieve cl(A) ***before*** it achieves on(B, C). Clearing A will entail removing C from A. Thus, when the system comes to achieving on(B, C), the status of block A will not be affected, since C will no longer be on top of it.



Action Sequence: MAKECLEAR(A), MAKECLEAR(B), PUTON(A, B)

dia.2.5 Holding periods after adding operator preconditions



Action Sequence: MAKECLEAR(A), PUTON(B, C), PUTON(A, B)

dia.2.6 Holding periods after goal promotion

The goal cl(A) has been promoted in such a way that the system will attempt to satisfy it before it tries to satisfy on(A, B), the goal that immediately preceded it. In this particular example, we come across a non-redundant sequence of operators very quickly.

2.2.5.3. INTERPLAN's Operation

INTERPLAN's principal data structure is called a "ticklist". This is a two dimensional array, the columns representing the individual goals in a conjunction of goals to be satisfied, and the rows representing world states in which goals are true. Each ticklist forms a node in a tree; an arc from a ticklist to an entry in a higher level ticklist represent the application of an operator. Goals in a high level ticklist that are to be protected can be added on as extra columns to lower level ticklists. INTERPLAN can perform a number of operations on the tree of ticklists, using the ticklist designated as current as a reference. It can scan a row of the current ticklist, to find if any of the facts are satisfied already in the world state, put "ticks" in for those that are, and "crosses" for those that are not. It can detect that a row in the current ticklist contains all successes (ticks), and backup to the next highest ticklist. It can reorder columns in the current ticklist, or add a column from the current ticklist to a higher level ticklist; these correspond to the reordering and the promotion of goals respectively.

These changes to the ticklist tree are made by what Tate calls *editors*. The particular editor to be used is chosen by a ticklist *classifier*, which classifies the current ticklist. INTERPLAN goes through cycles of classifying current ticklist, and editing ticklist tree; in the editing phase, the current ticklist can be changed, giving INTERPLAN the ability to grow a complete tree. Termination occurs when the ticklists on the leaves of the tree contain no crosses, only goals that are trivially true.

2.2.6. Waldinger's Goal Regression Method

Another planner for which the ability to tackle goal interactions was an important design criterion was a system devised by Waldinger ([WALDINGER, 1977]) which he called "goal regression".

In this system, plans are represented by two structures:

(1) A set, F, of operators, and

(2) A set of (protected) facts which must hold at given points in F.

set in one place, linear sequence in another

F is a linear sequence of operators, which if applied one by one to an initial world state, will transform it into a state in which a specified set of facts (the goals) are true. The goals are ~~be~~ presented to the planner as a conjunction. The system will first attempt to achieve them in the order specified. Having achieved the first goal, it will put it into the set of facts, so that it is *protected.* If a fact is protected, then any actions that the planner tries to insert that deny the protected fact are retracted. Facts may be protected at any stage in the plan; the goals are protected at the end of the plan. If the achievement of the goals in the initial order is not feasible, then the system tries other orders. If simple reordering is not adequate, then the system uses a means of resolving goal conflicts called *regression.* This works as follows. Supposing that we want a fact P to be true after the application of an operator f. This may only be possible if we make a fact P' true before applying f. To make P' true at this point, we may insert a new operator, or we may use further goal regression.

Waldinger used what he called a "skeleton model" to represent the world state. Instead of applying the operators to the world state as they are generated, the system simply stores the facts that will become true or false as a result of the operator being applied. That is, only the *changes* to the world state are stored. In order to find out if a fact, P, is true at some point in the plan, the system will first see if the operator at that point makes it true. If it doesn't, then an appropriate regression algorithm is used to find the fact P' that must hold *before* the operator for P to be true. The truth of P' is checked at this point. This process of regressing and checking is repeated, until a definite statement about the regressed value of P is found. This may not be until the initial world state is reached.

As an example, we may have a conjunction of two goals, P and Q, to satisfy. It may be possible to generate a set of operators, F1, to make P true, but any attempt to extend it to make Q true may violate the protection on P. We may then find that if we generate a set of operators, F2, to make Q true, and then extend it to make P true, the protection on Q is violated. So, the system will take the last operator from F1 (f1) and use a standard function to derive a fact Q' that needs to be made true *before* f1 to make Q true after it. The system will first check to see if Q' is already true at this point. If not, it will try to find an operator that will make it true, making sure that this operator does not cause any protected facts to be violated. If this, too, does not work, then the system will regress back over the last-but-one operator in F1, and so on, until a solution is found, or an attack on F2 becomes necessary.

2.2.7. PLANNER

We shall now look at a planning system that was based on procedural knowledge. This program was called PLANNER, and it was written by Carl Hewitt in the late 60's ([HEWITT, 1970]). Like several other similar systems, it was an attempt to provide a general framework for organising domain specific knowledge in a way usable to a planner. Three key features of this approach can be distinguished:

(1) Pattern directed procedure invocation

(2) Data bases

(3) Non-deterministic control structure

Information about the state of the world is kept as a set of assertions in a database. New assertions can be added and old ones erased during the planning process, to keep the database up to date. Information about operators (henceforward called theorems within this section, to agree with Hewitt's terminology) is entered by the user. Each theorem has a pattern associated with it. Theorems in PLANNER are not invoked by name; they are invoked by pattern matching.

Two types of theorem can be defined in PLANNER: those which are invoked as a direct consequence of an attempt to satisfy a goal statement (CONSEQUENT theorems), and those which are invoked as an attempt to change the database (ANTECEDENT and ERASING theorems).

An example of a definition of a CONSEQUENT theorem, using the robot world described in the section on STRIPS, could be as follows:

```
(DEFINE GONEXTO
  (CONSEQUENT (X) (NEXTO ROBOT $? X)
     (GOAL (OBJECT $? X))
     (GOAL (GOTO $? X) (USE MAKEGOTO)) ))
```

A number of points emerge from this. The first line opens the definition of a theorem, GONEXTO. This name is the action that the theorem represents. The next line specifies that the theorem is of type CONSEQUENT, uses one variable (X), and would be called by a pattern match to (NEXTO ROBOT $? X), where X is a variable ($? is a prefix used in PLANNER to denote variables). The third and fourth lines are effectively preconditions, which must be satisfied if the theorem is to succeed. The way a GOAL statement works is as follows. It first of all checks to see if it's argument already exists in the database; if so, it exits successfully. If not, it checks through the CONSEQUENT theorems that have been defined, to see if it can find one having a pattern matching it's argument. If so, it then attempts to execute the theorem; success of the theorem results in success of the GOAL statement. If the GOAL statement has a USE argument as well as a pattern argument, as in line four above, then the GOAL statement will be constrained to use only the theorem(s) specified. In the above, this would be the theorem MAKEGOTO. A USE without arguments specifies to the system that no theorems are to be used - only a check on the database is allowed.

PLANNER allows theorems to assert or erase facts from the database, using the ASSERT and ERASE procedures. When one of these procedures is invoked, an attempt is made to match the pattern being asserted/erased with the patterns of the set of ANTECEDENT/ERASING theorems. PLANNER will invoke every ANTECEDENT/ERASING theorem that it comes across with a matching pattern. This gives the user a way of defining the side effects of an effect, without explicitly coding it in every CONSEQUENT theorem definition. As an example, suppose, in the above robot world, that we wished to assert that every time an object X is next to an object Y at location Z, the object X is at location Z. We could use the theorem:

```
(THEOREM ADDNEXTO
  (ANTECEDENT (X Y Z) (NEXTO $? X $? Y)
      (GOAL (AT $? Y $? Z) (USE))
      (ASSERT (AT $? X $? Z)) ))
```

The format is much the same as that of the CONSEQUENT theorem defined above, except this is of type ANTECEDENT. In line 3 the GOAL procedure is actually being used to instantiate Z by matching (AT $? Y $? Z) with a fact in the database. The ASSERT then asserts the appropriate new fact in the database.

The user starts the planning process off by typing a statement such as

(GOAL (NEXTO BOX1 BOX2))

which is asking a robot to push BOX1 next to BOX2. Pattern matching will occur with an appropriate theorem, which may in turn invoke other theorems etc. Eventually, the chain of pattern matching, invocation and execution will terminate, and the complete plan is the sequence of actions (ie. theorem names) that makes up the chain. A conjunction of goals can be called by

(THPROG (<variable list>) (GOAL1) (GOAL2)......(GOALn))

The control structure used by PLANNER is based on backtracking augmented by an optional failure message system. Backtracking will occur when an attempt to satisfy a goal fails; this will cause the system to look at the last choice point, to see if any alternative theorems matching the given pattern can be found. If so, one of those theorems will be invoked, otherwise the choice point which gave rise to the current choice point will be tested in the same way, and so on, until an appropriate solution is found. If the user desires, it is possible to attach messages, in the form of patterns, to failures. These failures propagate backwards along the backtracking trail, until a matching "trap" pattern is found. The trap choice point is the node responsible for the failure; information carried with the pattern about why the failure occurred can be used to guide the selection of a more appropriate theorem.

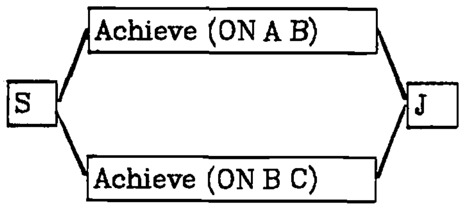
PLANNER also supports multiple databases, so that if too much effort is being expended on one particular theorem (ie. the search in the solution space is becoming too deep), then a switch to an alternative theorem with the same pattern can be made. If this fails, or becomes too expensive itself, then the system can switch back to the original theorem.

2.2.8. NOAH

NOAH was the first attempt to escape from the assumptions of linearity which characterised earlier planners. It was written by Sacerdoti in 1975 [SACERDOTI, 1975]. NOAH represents plans in a structure that Sacerdoti calls a **procedural net**. The nodes in this net correspond to goals to be achieved (which point to blocks of code to perform actions to satisfy the goals) and goals already achieved ("phantom" nodes). Links in the network impose a partial ordering on goals. When the planner is entered, the problem is represented in one node, as a conjunction of goals to be solved. The operators used by the planner are specified by a SOUP ("Semantics of User's Problem") code. This uses a LISP-like language to describe operators in terms of the new goals that they would insert into the net and the facts that they would deny. Preconditions are specified as tests on the facts that must hold before the operator can be applied.

For example, in the blocks world domain, we may wish to solve the problem (AND (ON A B) (ON B C)), with an initial world state ((CLEAR A) (CLEAR B) (CLEAR C)). NOAH will try to simulate the goal, and in so doing, will generate two parallel goals (see dia.2.7).

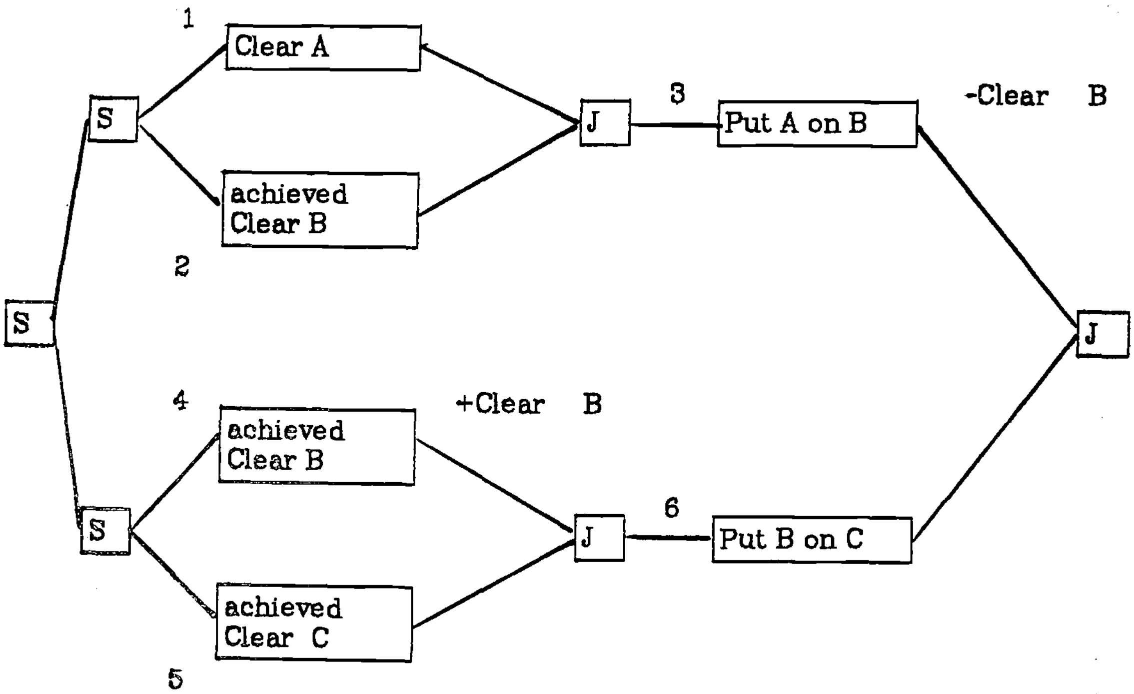
The boxes containing S and J are "split" and "join" nodes respectively. The other two boxes are the goals of the planner. Split and join nodes are used to enable the representation of parallel goals; parallelism implies that no ordering has yet been imposed. No assumptions about the ordering of the goals in dia.2.7 has been made, since there is not sufficient information available. Having performed this expansion, NOAH invokes a set of critics. These critics look at the expanded network to see if any conflicts have been introduced, and to see if any variables can be instantiated. Associated with every goal in the procedural net is an add and a delete list. These specify the



dia.2.7 Initial goals for blocks world problem

effects of the action associated with the goal, in terms of the facts which must be added to and deleted from the world model respectively. If an effect is introduced in one branch of the plan which negates an effect introduced in a parallel branch, then a conflict is said to have occurred. This can be illustrated if we carry on with the above example - further expansion produces the net in dia.2.8. [1]

The action (PUTON A B) at node 3 deletes the fact (CLEAR B), which is asserted at node 4. A conflict has thus arisen. To cure this, one of NOAH's critics, the RESOLVE CONFLICTS critic, will make a new link in the network,



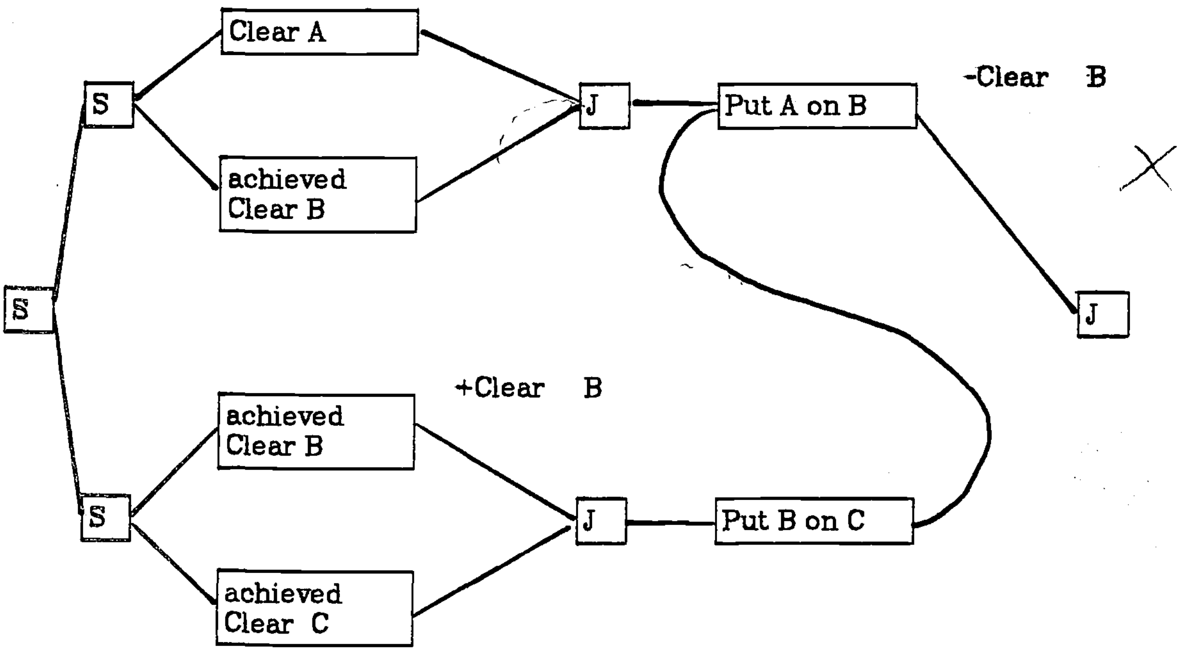
dia.2.8 Procedural net after expanding top-level goals

[1]Goals whose status is "achieved" are ones which have already been satisfied. The system must satisfy the remaining goals by, for instance, applying operators.

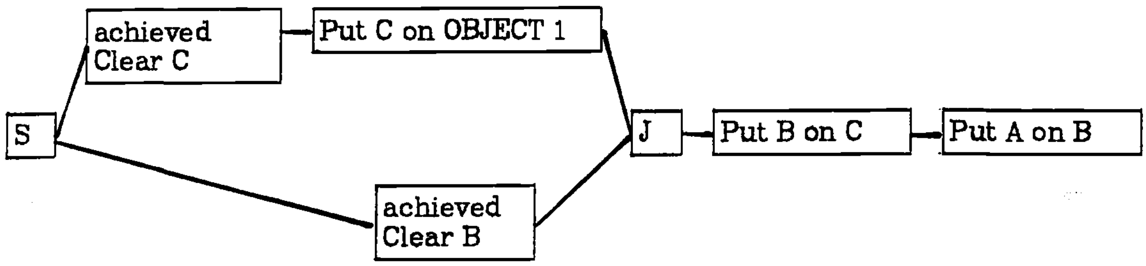
imposing a stricter ordering on the nodes (see dia.2.9).

Eventually, after cycling through several more expand/criticise phases, the final net is produced (see dia.2.10). In order to achieve "Clear A", the system has had to remove C from A, and put C on some arbitrary object, OBJECT 1. This created the goal "Clear C", which conflicted with the effects of the "Put B on C" action. Thus, the RESOLVE CONFLICTS critic suggested linking this action *after* "Put C on OBJECT 1". Redundant goal nodes were then removed from the net.

The operation of the planner can be summarised thus:



dia.2.9 Procedural net after correcting interaction



dia.2.10 Final Plan

(1) Attempt to expand all nodes in the current net. If no more can be expanded, then the planning process is complete.

(2) Criticise the new net, resolving conflicts, etc.

(3) Goto (1).

NOAH's control structure is rather inflexible, in that it makes no allowances for failure during planning. The assumption is, presumably, that the critics will always make the right decisions, so that failure can never occur.

2.2.9. Introduction to Tate's Work on NONLIN

NONLIN is, like NOAH, a non-linear planning program. It was originally developed in 1976 by Austin Tate [TATE, 1976]. It was written in POP-2, and ran on the Edinburgh DEC 10. It represents intermediate parts of the planning process as partially ordered networks of action and goal nodes. This is very similar in concept to Sacerdoti's procedural net, although NONLIN represents actions explicitly. The basic cycle of operation is similar to NOAH too; NONLIN expands all expandable nodes in the network, makes a check for interactions (conflicts, in Sacerdoti's terminology) after each expansion, correcting for any found, and then tries to find other expandable nodes. Hence the plan becomes more refined in detail, and more ordered, until, when no further expandable nodes can be found, a complete plan has been produced. NONLIN's control structure uses an augmented form of back-tracking to deal with plan time failures. The user can take advantage of this, since it enables NONLIN to generate alternative plans. A flow-diagram of NONLIN's operation is shown in dia.2.11. In this section, we shall look at the following features of NONLIN:

(1) Plan representation.

(2) Expansion algorithms.

(3) The "Task Formalism" - a means of describing the domain and the operators that change it.

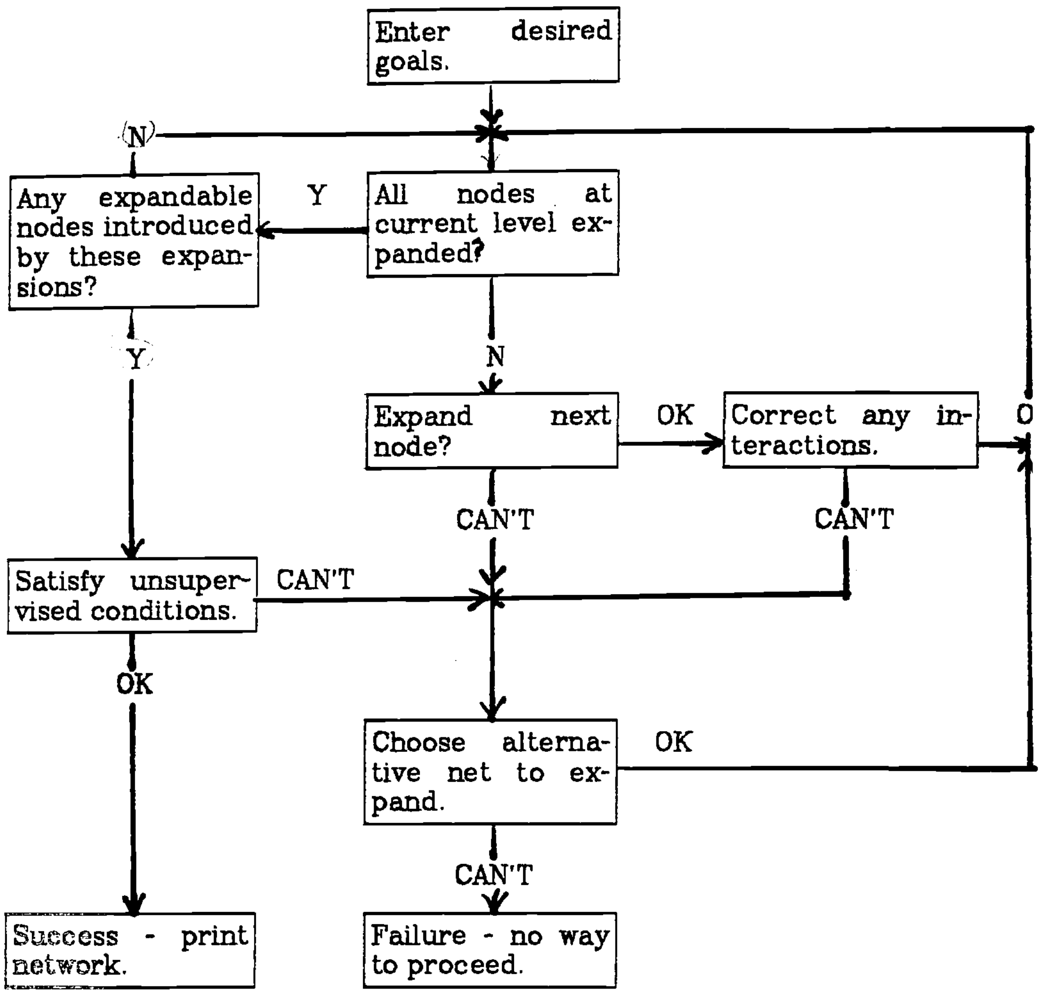
(4) The "QA system" - a means of finding out whether a fact is true at a given plan node.

(5) Linking processes in NONLIN - used to remove interactions.

2.2.9.1. Plan Representation

Three data structures are used to represent the plan:

(1) A list of the nodes in a network, called ALLNODES. Each element in this list contains information on the number of the node it represents, it's type (GOAL, PHANTOM, ACTION, DUMMY or PLANHEAD), a pattern, pointers to preceding and succeeding nodes, and a suggested expansion for that node. The plan network is a means of specifying the GOALs of the planner, and the ACTIONs that need to be performed to achieve them. It represents plans in a non-linear way; GOALs and ACTIONs can



dia.2.11 Flow diagram of NONLIN's operation

be in parallel, meaning that they could be arbitrarily ordered, without affecting the final world state.

(2) A list of the effects that arise as a result of executing operators. This is called the Table Of Multiple Effects (TOME), and each entry contains a pattern and value representing that effect, and the number of the node at which the effect arose. The TOME is used by the QA system (see later), to find out if a given pattern has a given value at a given node in

the plan network.

(3) A list of the conditions on nodes, called GOST ("Goal Structure"). Each element of this list specifies which patterns should have which values at which nodes, and also specifies the disjunctive set of nodes which contribute to the satisfaction of those conditions. The part of a plan network between a contributor node and the condition it satisfies is called a *range*. The GOST thus gives a set of ranges over which facts must b protected for the current plan to remain valid.

2.2.9.2. Expansion Algorithms

NONLIN allows 5 types of plan network node. Two of these, PLANHEAD and DUMMY, cannot be expanded. PLANHEAD is the node inserted into the plan network at the start of the planning process, and DUMMY nodes serve to join links, in rather the same way as the split and join nodes in NOAH. GOAL PHANTOM and ACTION nodes, however, are expandable:

(1) GOAL nodes are inserted where it is desired that their pattern should have a given value. There are three ways in which this can be done:

a) The pattern already has the value at that point

b) The pattern can be given the value by linking

c) The pattern can be given the value by expanding the node.

If a) or b) succeed, the node will be returned with type PHANTOM.

(2) PHANTOM nodes hold patterns which are assumed to have been given a value already. If, due to later plan changes, the pattern is found not to have this value, the node type is changed to GOAL, and a goal expansion is performed as above.

(3) ACTION nodes are assumed to be present as commands to do something. During expansion, no checks are made to see if their patterns are true or not; they are expanded as in (1) c). If the action chosen is primitive, no expansion occurs, although effects may be put into TOME, and conditions into GOST.

If an expansion inserts new nodes into the network, links ***from*** previous nodes and *to* successor nodes are made for the first and last nodes of the expansion respectively (the list of previous nodes will be referred to as the "prenode list", and the successors will be referred to as the "succnode list"). Similarly, any conditions from the higher level node transferred to the first node of the expansion, and any effects are transferred to the last. All the nodes in the expansion are tagged onto the end of ALLNODES, except for the last, which replaces the original higher level node.

2.2.9.3. The Task Formalism

The Task Formalism is a means of specifying the operators which the system will use, and of describing the domain in which planning will occur. It is intended to be fairly general. An example of an operator specification is shown in DP 12 . This is a "blocks world" operator, which suggests what to do if the goal "clear the top of block X" is to be achieved. When selecting this operator, the system will have matched a GOAL in the plan network with the PATTERN specified in line 2. Before applying the operator, the CONDITIONS will be checked; they specify that there must be something on X, and that there must be a clear space on Z for dumping the excess block. There are 3 types of condition:

```
OPSCHEMA MAKECLEAR
            PATTERN {CLEARTOP $*X}
            EXPANSION  1  GOAL   {CLEARTOP $*Y}
                       2  ACTION {PUT $*Y ON TOP OF $*Z}
            ORDERINGS 1—>2
            CONDITIONS HOLDS {ON $*Y $*X} AT 2
                       HOLDS {CLEARTOP $*Z} AT 2
            VARS  X <: NON TABLE: >
                  Y UNDEF
                  Z <: ET <: NON $*X: > <: NON $*Y: >: >;
END;
```

dia.2.12 Example NONLIN OPSCHEMA

(1) If a HOLDS[2] condition occurs, then the fact specified in the condition *must* be true before the system will consider applying the operator.

(2) SUPERVISED - these conditions are satisfied by events brought about by the operator itself. If the user so desires, SUPERVISED conditions can be inserted automatically; there will be one for each new GOAL node introduced by the operator.

(3) UNSUPERVISED - these need not be satisfied immediately, and in general, are satisfied when all of the other planning at a given level in the abstraction hierarchy has been completed.

It is the conditions which order the plan; as the plan evolves, it becomes more ordered due to the addition of further conditions. The plan itself is a partially ordered network of nodes, the nodes corresponding to either goals or actions.

When an operator is applied, any GOAL or ACTION nodes in the EXPANSION part are substituted for the original node in the network. The example

[2]"USEWHEN" is used in Tate's more recent papers, but the older terminology is retained here, because it will appear later on in the description of MODPLAN's operation.

operator will introduce 2 new nodes, a GOAL node, specifying that block Y should be clear, and an ACTION node, which will put block Y on top of block Z.

The facts about the domain are represented by two definitions, called ALWAYSCTXT and INITCTXT. ALWAYSCTXT contains a list of the facts that will always be true. These facts cannot be changed by the planning process, but will affect it's direction. INITCTXT contains a list of the facts that are true in the initial world state; some or all of these facts may be affected by planning.

2.2.9.4. The QA System

The QA (Question Answering) system is intended to answer questions about the value of a given pattern at a given node in the network. It is used to check nodes before they are expanded, as already explained, and it is also used to suggest links for the satisfaction of UNSUPERVISED conditions. It is not possible to define the world state at any particular point in the planning process, because the network is only partially ordered. For instance, we could not say which of two parallel nodes came first, chronologically. So rather than trying to store world models at every point, only *changes* produced by effects are stored.

The "patterns" discussed in this section correspond to facts, such as ON(A B). Patterns will generally be denoted by P, their values by V, and a node in the plan network by it's number, N. In the case of ON(A, B), the pattern can only have two values, namely true or false. However, the QA system is capable of handling patterns with any number of possible values, and in this context, "not-V" means any value other than V.

The QA system helps to answer two types of question:

(1) Does a pattern P have a value V at a node N in the network, and

(2) If not, what links would have to be made to make P have this value at N?

The QA system divides nodes relevant to the question asked into two classes - V nodes, at which P is given a value V, and not-V nodes, at which P is given a value other than V. The QA system's reply comes in the form of four lists of nodes:

(1) **Critical V nodes.** All V nodes linked before N for which there are no intervening V nodes or not-V nodes.

(2) **Critical not-V nodes.** All not-V nodes linked before N for which there are no intervening V nodes or not-V nodes.

(3) **Parallel V nodes.** All V nodes in parallel with N.

(4) **Parallel not-V nodes.** All not-V nodes in parallel with N.

The QA system also stores the facts about the world which are always true and which were true in the initial state, in ALWAYCTXT and INITCTXT respectively. When the system is looking for a fact, it will look in ALWAYCTXT first, and then in the network, to improve efficiency (assuming that the bulk of the information about the world is in INITCTXT and ALWAYCTXT, and that changes due to the plan have a relatively small effect). *

2.2.9.5. Linking Processes

There are two occasions when it is necessary to put new links into the network:

(1) When linearisation is required to remove interactions. TOME entries are of the form

* This is not correct, c.f. pp. 23 of DAI report.

{<pattern> <node number> <value of pattern>}

When such an entry is made (usually by an operator), a check is made to see if

a) any parallel node has an effect that makes the value of the pattern different (found from TOME), and

b) There are any parallel ranges with the same pattern but an opposite value (found from 'Gost').

If either a) or b) are true, then *interactions* are said to have occurred. Once the expansion has been completed, an attempt can be made to correct for these interactions by making links in the network to order the interacting ranges of nodes either before or after the effects intro duced.

(2) When it is desired to make a pattern P have a value V at a node N. The four lists of critical nodes generated by the QA system are used to suggest ways of doing this. The procedure is to link in one critical V node before N, if such a link does not already exist. This may suggest one or more networks, and NONLIN must link-out all critical not-V nodes from between a linked-in critical V node and N. The terminology used should become clearer once the QA system has been explained. The way in which the system actually makes the links in this case is to generate an interaction record, and then allow the interaction correction system make appropriate links.

The interaction correction procedure works as follows. Given a list of pairs of interacting ranges, the system will attempt to eliminate the cause of the interaction for each one in turn. There are seven possible ways of correcting interactions, some of which might not be applicable in all cases:

(1) Link the first range before the second.

(2) Link the second range before the first.

(3) Link the first range before the second, breaking the second range.

(4) Link the first range before the second, breaking the first range.

(5) Link the second range before the first, breaking the second range.

(6) Link the second range before the first, breaking the first range.

(7) Break both the first and the second ranges.

"Breaking" a range means removing the first node in the range as a contributor to the last. In many cases, this may not be possible, eg. if the first node is the only contributor to a HOLDS condition, in which case, only the first two interaction correction methods are viable. There are restrictions on linking too. For instance, it is not permissible to put a link into the net that will give rise to a loop. All of these interaction correction techniques are attempts to make sure that effects that arise in one part of the net do not interfere with the satisfaction of conditions in other parts of the network.

2.2.10. The Decision Graph in Plan Generation and Execution

2.2.10.1. General Introduction

A decision graph is a way of augmenting backtracking so that it is ***dependency directed***. Many planning systems employ backtracking as the basis of their control structure. If it is possible to represent the kinds of decision made during planning explicitly, then it ought to be possible to build up a graph during the planning process, specifying the decisions made and how they depend on each other. Failures can occur either during plan generation or execution. Normal backtracking would destroy the effects of all decisions in a chronological sequence from the point of failure to the decision deemed responsible for the failure. Given the dependency information stored in the graph, it should be possible to *preserve* the effects of all decisions not dependent on the decision responsible for the failure. Thus, we save as much of the original planning effort as possible. Below, two pieces of work are reviewed in detail, both done at Edinburgh. The first is the research of Philip Hayes, on the use of a decision graph in the generation of revised plans after plan execution failure. The second is a later piece of research, carried out by Lesely Daniel, in which a decision graph is used in non-linear plan generation. Other work on dependency directed backtracking has been used for "reason maintenance" (eg. Stallman and Sussman [STALLMAN, 1977], Doyle [DOYLE, 1976]).

2.2.10.2. P. J.Hayes' Work

2.2.10.2.1. Purpose and General Description

2.2.10.2.1.1. Introduction

Hayes came to the conclusion that all planning could be described in terms of subgoaling and refinement. Subgoaling is the creation of two or more goals to replace a single goal, where the subgoals are more primitive than the goal that they are replacing. Refinement is the addition of more detail to an existing goal to produce a more refined goal. Refinement could be regarded as subgoaling with only one subgoal, since the result of refinement is always to make a goal more primitive.

Hayes devised a representation for plans that consisted of two parts:

(1) A plan tree, in which the successive applications of subgoaling and refinement are stored.

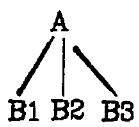
(2) A decision graph, in which the decisions producing each instance of subgoaling and refinement are logged, along with information about the way decisions influence each other.

The information about decision dependency is used to make replaning on execution failure more efficient; it allows the system to discard only that part of the plan directly dependent on the decision responsible for failure. Replanning can start, at an entry point decided on by the system, from the remaining plan.

2.2.10.2.1.2. The Plan Tree

Nodes in the plan tree represent goals and their associated actions. Arcs represent the parent/child relationship. The topmost, or root, node, corresponds to the overall goal, towards which planning is aimed.

Subgoaling is represented by a split in the tree, eg. if goal A is subgoaled into goals B1, B2 and B3, this would be represented as



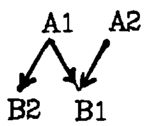
Refinement is simply represented by a single link, eg. if A is refined to B, we would get

The tips of the complete plan tree correspond to the primitive actions required to achieve the top level goal. In general, the further down a node is in the plan tree, the more detailed it is. However, there is no restriction in the order in which goals may be subgoaled or refined, and one (perhaps critical) part of the plan could be developed to a greater depth than other parts, if so desired.

All nodes in the plan tree have links to their parent and child nodes, and to the decision graph nodes responsible for introducing them. Plan graph nodes will be referred to as j-nodes, after the "journey" nodes used in Hayes' travel planning system.

2.2.10.2.1.3. The Decision Graph

The decision graph consists of a network of nodes (d-nodes) which correspond to the decisions made in the planning process (ie., subgoaling and refinement decisions). The arcs connecting the decisions are directed, and indicate the dependency of one decision on another, eg., in the diagram below, B1 is dependent on A1 and A2, and A1 has two nodes dependent on it, B1 and B2.



Each d-node is directly responsible for the introduction of one or more j-nodes, and has pointers to them. A d-node is indirectly responsible for any j-node introduced by one of it's ancestors.

In Hayes' system, a failure to perform a j-node action causes the d-node *directly* responsible for the introduction of this j-node to be found. The system will use this as a re-entry point into the planning process, but before this can be done, the part of the plan invalidated by the failure must be removed. This is achieved by UNDOing the d-node identified as being responsible for the failure, and works as follows:

(1) Remove any j-nodes introduced by the d-node into the plan tree.

(2) Remove the d-node from the decision graph.

(3) UNDO all of the child nodes of the d-node in the decision graph.

Any decisions which are **not** dependent on the d-node so UNDOne, even though they may have been introduced after it, will not be affected. Thus, the parts of the plan introduced by these independent decisions will be preserved. This is in contrast to a simple backtracking algorithm, in which the effects of all decisions on the plan would be removed in reverse chronological order, until the decision responsible for the failure had been arrived at. This would occur irrespective of whether the decisions so removed were dependent on the failing decision.

2.2.10.2.1.4. Plan Execution

If plan execution fails, the execution monitoring system will designate some set, J, of j-nodes as unexecutable. The j-nodes so designated would be the deepest possible, ie. the nearest to the tips of the plan tree, in order that as few decisions as possible are affected when invalidated parts of the plan are removed. Given this set of nodes, then, the following steps are taken:

(1)Any parts of the plan already executed are discarded, along with any d-nodes responsible only for j-nodes in the discarded parts of the plan.

(2)Any information in the plan made inaccurate as a result of the failure is discarded.

(3)For each j-node, j, in the set J, do the following:

(a)If j was discarded by (1), then do nothing, else

(b)UNDO the d-node, d, responsible for the introduction of j

(c)Enter the planner via the entry point associated with d

(d)If replanning succeeds, do nothing more, else

(e)If j is not the node of the plan tree, reset j to it's parent node, and go back to (b), else

(f)Stop, because the failure has made attainment of the top-level goal impossible.

In replanning, the decision supplying the entry point must be remade. It may also be necessary to make independent decisions. For instance, supposing the failure occurred during subgoaling. In choosing the most appropriate set of subgoals to insert, we may find that, due to the circumstances of the failure, some of the conditions on the subgoals are not satisfied. Hence, we may want a decision, that will make these conditions true, to be inserted into the decision graph; this decision would not depend on the decision to subgoal.

2.2.10.2.2. The Travel System

2.2.10.2.2.1. Introduction

To illustrate the use of the decision graph (and of domain dependent knowledge), Hayes chose a travel system as his domain. The program was capable of planning journeys, by rail, sea or air, within Europe. It had facilities for imposing constraints on total cost and time of journeys, and arrival times at particular places.

There are a number of features which are specific to the journey domain. All decision graph nodes have associated with them a function, called the dfunction. This is the function which made the decision, eg. **train-time** decides which service on a particular train route to use. Also associated with each d-node is a variable called CURRENT, which stores the number of the j-node which the dfunction is to operate on. If replanning occurs, the dfunction is used as the entry point to the planner, using CURRENT as the j-node to start from.

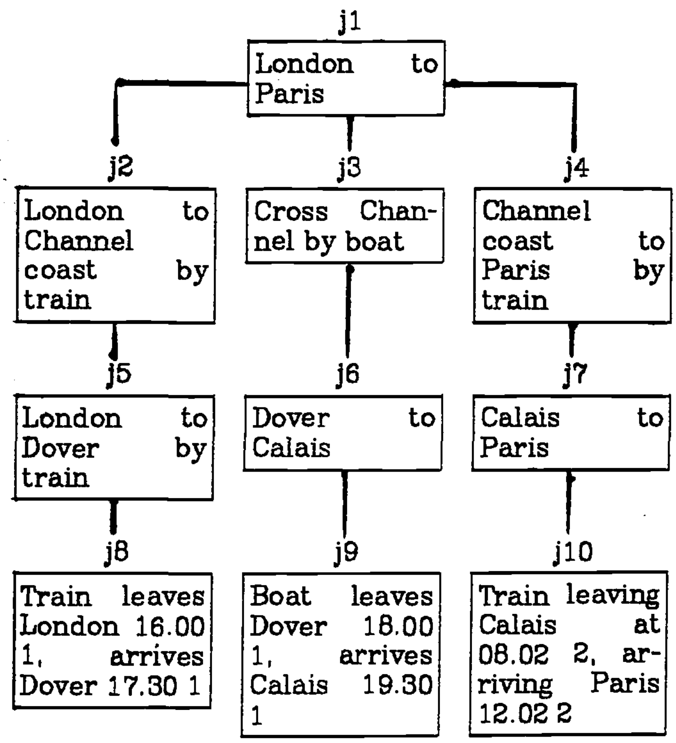
One problem with using the decision graph is this: what happens if all possible alternatives have been exhausted at a particular decision point, and that point has more than one parent node? How does the system choose the most appropriate parent to try? Hayes gave no general solution to this, but in the case of the travel system, this is set by the designer, who states certain types of decision to be preferable to others. Eg. the system will try to change a timing decision before trying a routing decision.

The next section gives an example of how the system might react to a failure during execution. It is a condensed version of one of the problems given in Hayes' thesis. It will be seen from the example that the goals are of type "journey" only, nodes deeper in the tree simply representing more fully specified journeys. Arrival and departure times are given terms of a 24 hour clock time, followed by the day. Eg., 16.00 2 means 4-o'-clock in the afternoon on day two of the journey.

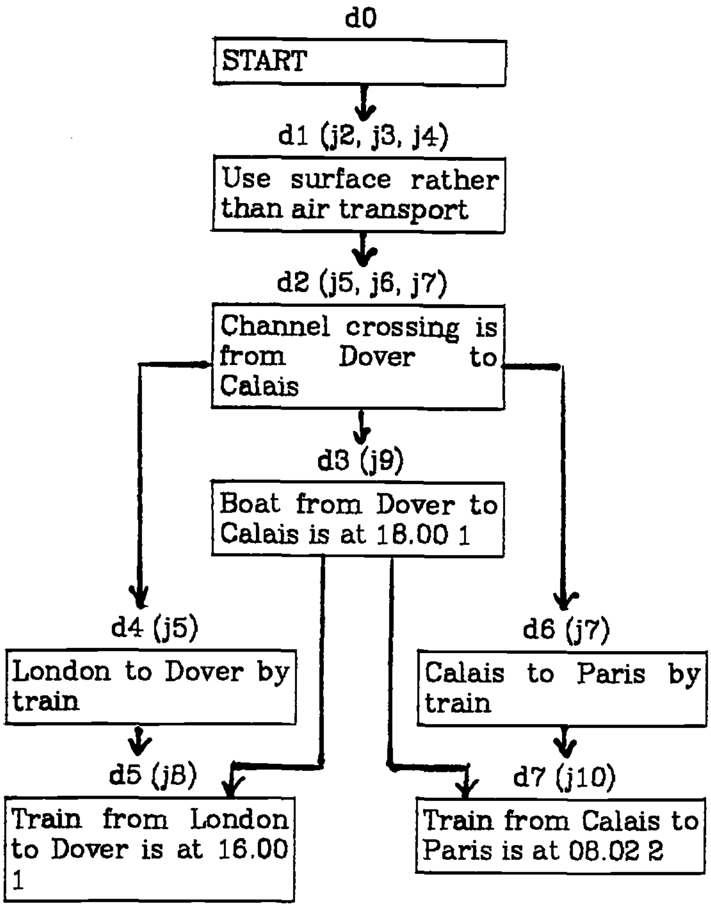
2.2.10.2.2.2. The Problem: A Journey From London to Paris

The system is told to plan a journey from London to Paris, with the constraint that air travel is not to be used. The plan shown in dia.2.13 is generated (the numbers on it are the j-node numbers). Each level in the diagram represents greater detail in subgoal description.

The decision graph associated with the plan is shown in dia.2.14(the numbers on it are the d-node numbers; the bracketed numbers refer to the j-nodes inserted)



dia.2.13 Plan tree for travel problem



dia.2.14 Decision graph for travel problem

Execution of the plan corresponds to moving from left to right across the tips of the plan tree.

Supposing, however, that an unexpected event occurred before we had had a chance to complete execution of the plan - the train from London to Dover is cancelled. The j-nodes which failed, j8, j9 and j10, were introduced by d-nodes d3, d5 and d7 respectively. These must, therefore, be UNDOne. This leads to the removal of j-nodes j8, j9 and j10 from the plan. The planner is now restarted via the entry point of d-node d3, for which j-node j6 is current. A new set of j-nodes, j11, j12 and j13, replace j-nodes j8, j9 and j10, respectively. These nodes are as follows:

j-node 11 Train leaving London 16.30 1 arriving Dover 18.00 1

j-node 12 Boat leaving Dover 19.00 1 arriving Calais 19.30 1

j-node 13 Train leaving Calais 21.00 1 arriving Paris 21.42 1

No further unexpected events arise, and this new plan enables the system to complete the journey successfully.

2.2.10.3. Daniel's Work on NONLIN With a Decision Graph

2.2.10.3.1. Introduction

Hayes used the decision graph to make replanning on execution failure efficient. However, possibly following a suggestion in Hayes' thesis, Daniel decided to use the decision graph as a means of logging the decisions made, and their interdependency, during plan generation, with the aim of providing a better control structure than backtracking. Daniel based her work on Tate's NONLIN, to which she added the decision graph.

As well as adding the decision graph to NONLIN, Daniel also gave the user the ability to impose global constraints on the cost and time of execution of the plan. Each possible choice of action has associated with it information about it's cost and duration, whilst goal and phantom nodes are assumed to have zero cost and duration. Hence, for a plan at any given level of detail, overall time and cost can be estimated. Choices of operator for a given expansion can be made in the context of minimising a function relating overall cost and duration of the plan.

The way that the system works is similar to Hayes' method. If a plan-time failure condition is detected, the decision responsible for the failure is identified by built-in heuristics. This decision is undone, and the planning process is then rentered at some predetermined point, from which normal planning can continue. The operation of the system is intended to be completely general, and no domain dependent heuristics are used. However, the user can influence the system to some extent, by imposing global cost and time constraints.

2.2.10.3.2. The Decision Graph

The first step in implementing a decision graph would probably have been to identify the kinds of decision made by NONLIN during the planning process. Daniel identified three kinds:

(1) Choice of expansion method for a node

(2) Introduce a link to make a goal node into a phantom

(3) Choice of linearisation for removing an interaction

The three different types of node in the graph reflect these different decision types. Each node in the graph corresponds to a decision made, and contains information on the decision type and number, plus a list of pointers to the network nodes affected, and dependent decisions. These nodes are called d-nodes, as in Hayes. The different d-nodes are set out as follows:

(1) Expansion node

netnode   Pointer to plan network node being expanded newnodes — A list of pairs, each pair containing pointer to a plan network node introduced by the expansion, plus a pointer to the d-nodes responsible for expanding the plan network node, if any.

phantom links — List of pointers to any d-nodes corresponding to phantom links using a pattern achieved by the expansion.

interactions — List of pointers to any interaction d-nodes dependent on this expansion.

(2) Phantom Link Node

parent — A pair, of which the first points to a plan network node made into a phantom by linking, and the second to the corresponding d-node.

child — A pair, of which the first element points to the node establishing the goal of the above plan network node, and the second to it's d-node.

(3) Interaction Node

link — A pair, of which the first points to the plan network node at the beginning of the link introduced to remove the interaction, and the second to the end.

needed — A pair, pointing to the plan network node whose condition is involved in the interaction, and it's d-node.

added — A pair, pointing to the plan network node which achieved the above condition, and it's d-node.

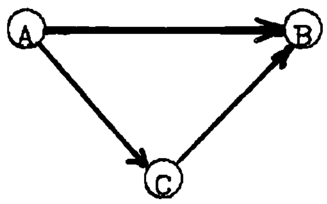
deleted — A pair, which points to the plan network node deleting the above condition, plus it's associated d-node.

2.2.10.3.3. Failure and Replanning

Daniel isolated two types of failure: infeasible networks and inefficient networks. In both cases, the state of the plan is examined, and an attempt is made to locate and correct the cause of the failure.

2.2.10.3.3.1. Infeasible Networks

These manifest themselves as irresolvable interactions of the form



where a condition on B requires an effect from A, but this effect is deleted by C.Daniel defines the action to be taken on detecting such an interaction in terms of the type of the condition on B. For instance, if it was a HOLDS condition, then the action to be taken is to undo the expansion introducing it. Presumably, since the expansion was identified as being responsible for the failure, replanning starts by making another expansion choice for this node.

Undoing an expansion node is similar in concept to the UNDO of Hayes, although somewhat more complicated, owing to the greater richness of the NONLIN representation for goals and actions. It involves removing nodes and links introduced by an expansion d-node and it's descendants from the network, re-attaching all links from removed nodes to the parent, and updating TOME and GOST. Interaction d-nodes are never *directly* undone; they are undone as a consequence of undoing an expansion d-node.

2.2.10.3.3.2. Inefficient Networks

At any time in the planning process, a check can be made on the overall estimated cost and duration for the plan. If either exceed a given limit, then the network is defined as inefficient, and action is taken.

In the case of the cost of execution becoming too high, the system will select jobs with large slack times, and replace them with cheaper alternatives. The slack time of a job is the difference between the time *available* for doing a job, and the time it *actually* takes to do the job. Replacing a job would presumably be done by undoing the decision responsible for inserting the job, and then restarting the planning process at the node involved, making an operator choice within some calculated constraints on cost.

If the total duration becomes too high, the system will replace critical jobs (those with a small slack time) with faster jobs. The replanning mechanism is probably similar to that mentioned above, using constraints on *time* rather than on cost.

2.2.10.4. Summary

Philip Hayes developed a structure, the decision graph, for logging decisions made in the planning process, and recording the dependency of one decision on another. His main aim in this was to enable a plan execution system to preserve the maximum amount of original planning effort, should an execution failure occur. If such a failure did occur, the parts of the plan already executed were removed, as were any decisions dependent on the decision responsible for failure, and plan nodes introduced by these decisions. Replanning then commenced from a pre-established entry point, and, once completed, execution would resume.

The representation devised by Hayes for plans was intended to give the user flexibility in the incorporation of domain dependent decision making functions, using domain knowledge, and the introduction of dependency links into new d-nodes.

Daniels' use of the decision graph was to make replanning on plan-time failures more efficient. Although she used three different types of decision graph node, rather than only one, the way in which dependency links could be made between these nodes was fixed; domain dependent knowledge was not used. Similarly, no domain dependent heuristics were allowed in the detection of failure conditions, or in choosing resumption points for planning. However, the system almost certainly afforded a considerable improvement over backtracking.

2.2.11. McDermott's NASL

2.2.11.1. Introduction

McDermott evolved a theory of problem solving, and applied it to a program called NASL, which designed electronic circuits. In the theory, problem solving is part of a more general class, that of actions. Solving a problem involves generating a plan and executing it.

McDermott outlined five goals for a theory of problem solving; in NASL, he attempted to tackle (2) and (3):

(1) **Power.** The more solution techniques available to a problem solver, the more powerful it is.

(2) **Analytical adequacy.** The notation used by the theory must encompass a large number of concepts relevant to it, eg. plans, events, etc.

(3) **Heuristic adequacy.** A program implemented to embody the problem solving theory must be able to accept facts in the notation mentioned above. The degree to which the program is heuristically adequate depends on it's ability to use these facts.

(4) **Application independence.** A theory of problem solving should be independent of any application that it may have.

(5) **Empirical veracity.** If the theory is intended to explain human problem solving behaviour, then it must be based on studies of how humans perform.

2.2.11.2. Theory

A **problem** is a **task** whose **action** cannot be immediately executed. A problem is solved by breaking it into **sub-problems** repeatedly, until all sub-problems become trivial. The sequence of such trivial sub-problems is called a **plan**, which, if executed, will bring about the solution of the top level problem.

A task is classified by three indices:

(1) **Problemacity.** A problem can be either **primitive** or **problematic.** In the first case, it is trivial, and does not need any problem solving effort to reduce it. In the second case, it needs to be broken into sub-problems, the sub-problems then being tackled as problems in their own right.

(2) **Monasticism.** If the solution to the problem does not entail changes to the "real world" (ie. the solution is conceptual), the problem is **inferential.** Otherwise, it is **worldly.**

(3) **Parasitism.** The **primary** actions of a task may be accomplished in isolation, but the **secondary** tasks (also referred to as **policies**) are carried

out in conjunction with primary tasks.

In producing a plan to solve a problem, the problem solver should, for each of the sub-tasks comprising the main problem, find a **sub-problem schemata**. A plan is an indexed collection of such schemata, each of which specifies how a particular sub-problem should be reduced. During planning, it is often necessary to decide which schema to use if more than one exists for a particular problem. It would be possible to maintain a trail of the system's choices, and to use backtracking to explore all possible methods for solving problems. McDermott, however, opted for a different strategy. No record is kept of the choices made by the system; if a choice is made, it is assumed to be the correct choice. Whenever the system discovers that a problem has become primitive, it is executed.

Choices are made by **choice rules**, which are selected on the merits of the situation at hand. One common situation is that other tasks could interfere with the current one. Choices taking into account scheduling and resources would have to be made at such points.

Errors may occur during planning, if a problem reduction method cannot be found for a problem, or during execution, if executing a task fails. In the first case, the system, not having any record of previous problem solving efforts, will attempt to **restate** the problem, and solve it in a different way. Execution error will cause the rectification of the failure to be treated as a task. The original (primitive) sub-problem leading to this failure will be replaced by the problem of rectifying the failure.

2.2.11.3. Implementation

Knowledge about problem solving and the domain in which problems are to be solved is kept in a database. Objects in the database are represented in a LISP-like language. A theorem prover is used to manage this database, and overall control of the system is in the hands of an interpreter. The interpreter goes through cycles, in which it selects (at random) a task, executes or reduces it (depending on whether the task is primitive or problematic), and then returns to select another task. Termination occurs when there are not more tasks left.

A task is an object in the database, and has associated with it a name, input and output variables, and an action. For instance, a task to find someone ("sucker-1") who might give away $100 could be represented as:

```
(TASK FIND-SUCKER <>
      (LAMBDA () (FIND (LAMBDA (X)
                       (? X would be willing to give me $100))))
      <'(SUCKER-1)>)
```

There are two kinds of relationship between tasks - **subtask** and **successor**. If our overall task is "get-money", then "find-sucker" could be one of the subtasks required to achieve it. This could be expressed in the following database statement:

```
(SUBTASK FIND-SUCKER GET-MONEY)
```

Having found a sucker, the next task is to ask him for money. This could be performed by a "squeeze" task; this is the next stage, or the successor, in our plan to get money. In the database, the successor relationship is expressed thus:

```
(SUCCESSOR FIND-SUCKER SQUEEZE)
```

Each task in the database has associated with it an enablement status; only if this status is "enabled" can this task be selected by the interpreter for attention. A task becomes "enabled" once all of the tasks preceding it have been completed, and all of it's supertasks have status "subs-enabled".

The way that the interpreter deals with a task depends on whether it is primitive or problematic. If it is primitive, then it is executed. There are two types of primitive action, and these are dealt with differently:

(1) **Built-in**. These are LISP functions, such as "do-subnet", which is used to reference "canned" plans.

(2) **Model manipulation axioms**. These are specified by the user for the domain, and indicate the propositions that are added and deleted by the action.

If the interpreter finds that a task is problematic, it makes the theorem prover prove a "to-do" theorem, to find ways of reducing the task. If only one way of reducing the task is found, then this is made into the principal sub-task of the current task. If more or less than one way of reducing the task is found, then the interpreter enters a "choice" or "rephrasing" protocol respectively.

The "choice" protocol is entered if there are several possible ways of reducing a task. The fact that a choice is to be made for this particular task is recorded in the database, and then, for each possible choice, an "option" object is created in the database.

For each of these options, the system will seek advice, which is expressed in terms of the options that should be ruled-out, ruled-in, and combined. If, after applying the advice for all the options, there is still more than one option left, then the system will fall back on user supplied selection rules.

If no ways of reducing a task are found, then the task of "rephrasing" the failing task is created. This is performed by a set of user supplied rules. These find a different representation for the failing task, and put it into the database as a new task to be reduced.

Policies are secondary tasks, that influence the reduction of some primary task. There are two ways in which they can do this - the user can define them, using "task" formulas in the database, or they can be primitive. Nonprimitive policies can influence primary actions by putting more tasks in the database to be done, and by changing the way in which choices are ruled-in and ruled-out for the primary task. The only important primitive policy in NASL is "monitor". This policy waits for a task to remove a formula from the database. If such an event occurs, a new task is created, in which the task removing the formula is recorded.

A nonprimitive policy is defined in the same way as a primary task, but two extra formulas are needed. First, the system is told that a given task is secondary by the formula

(POLICY task-name action).

Second, the primary tasks influenced by the policy are indicated by "scope" formulas:

(SCOPE secondary-task-name primary-task-name).

The "action" in the "policy" formula provides a means of executing the steps of the policy, and finishing it.

2.2.12. Stefik's MOLGEN

We will be concerned mainly with the layered control structure of MOLGEN here, rather than it's ability to handle constraints or plan genetics experiments. For more details on all aspects of MOLGEN, refer to Stefik, [STEFIK, 1981a, b] and Martin et al, [MARTIN, 1983]. Stefik argues that, because normal planners mix decision making about what to do in the domain being modelled with decision making about what to do to the plan itself, the latter considerations will manifest themselves in an inconsistent way at the domain level. What is more, such systems are not able to make sense out of general advice, such as "expand easily satisfied goal nodes first". The solution proposed is a *multi-level control structure* , with a planner at each level scheduling actions on the level below. MOLGEN has three such levels (or *spaces*). The top level, the *strategy space* is controlled by a simple interpreter. This governs the broad planning strategy, using either a least commitment or heuristic approach. Below the strategy space, and directly controlled by it, is the *design space* , which knows how to design plans. At the bottom of the heap is the *domain space* , in which plan steps are actually executed.

Each space has a hierarchical representation, and it is possible to define objects and operators with varying degrees of abstraction. For instance, in the genetics domain, the program has a very general **Merge** operator. A specialisation of this operator is the **Ligate** operator, which joins together strands of DNA.

The operators in MOLGEN's design space fall into three classes. The first is *comparison operators*, which compare goals and find differences. These differences are used to set up further goals and domain operators. The second class is what Stefik calls *temporal-extension operators*. These use the differences produced by the comparison operators to add more goals and operators at the current abstraction level of the plan. Lastly, there are a number of *specialisation operators*, which make the current plan more detailed, by specialising objects and operators, and generating and propagating constraints. The design space corresponds roughly to Wilensky's meta-planning level (see next section). At any stage, if an operator in the design space cannot complete it's job (eg. because a variable is not sufficiently constrained), the job can be suspended. Suspended design steps are marked, so that they can be restarted at some later time.

MOLGEN's strategy space only has four operators, **Focus**, **Resume**, **Guess** and **Undo**. The first two are used for running the system under a least commitment régime. The second two allow the system to run under a heuristic régime. The interpreter decides which régime the system should be running under, and in fact, simply switches from least commitment (the normal régime) to heuristic, if the least commitment approach should fail.

The strategy space operators work as follows. **Focus** asks all of the design operators if there are any jobs that they could do at the design level. The jobs found are put onto an agenda of tasks. These tasks are executed sequentially, **Focus** being called after each execution, adding new tasks to the agenda each time. If a task cannot be executed, control is passed back to the top-level interpreter. **Resume** reactivates dormant design steps, and attempts to carry each task on it's agenda as far as possible towards completion. If **Focus** and **Resume** run out of things to do, then **Guess** will force each of the operators for suspended design steps to commit themselves to a particular option. Control is then passed, via the interpreter, to **Focus** again. **Undo** forces the system to backtrack if a plan becomes over-constrained.

At each level in the control structure, there is an interface to the level below, in the form of a message passing program. This enables decisions made at one level to be implemented at the level below.

2.2.13. Work of Wilensky and Faletti on PANDORA

Wilensky's main complaint about current planners is that they contain no explicit knowledge about *how* to plan. His favourite example is the detection and correction of goal interactions. He presents Sussman's HACKER ([SUSSMAN, 1975]) as a particularly bad instance of non-explicitness in representing "how to plan" information. HACKER has a set of critics, which contain procedural knowledge about how interactions may be avoided etc. Because this knowledge is locked up within critics, it cannot be manipulated or shared by other critics. Wilensky's argument is that this sort of information should be represented as "meta-goals", and that the job done by critics be replaced by "meta-planning", using the same (general) problem solver that constructs normal plans.

Wilensky's notation for the motivations of the planner and meta-planner assume that the system has a collection of "themes" and "meta-themes", which give rise to specific goals and meta-goals respectively. The particula theme or meta-theme selected depends on the situation.

A list of Wilensky's meta-themes is given in table.2.1, showing the situations under which they might be activated, and the meta-goals that would be initiated.

A planner, PANDORA, was implemented by Wilensky, embodying the concepts of theme and meta-theme. PANDORA has three major components. The first is a **Goal Detector.** It's function is to examine the state of the world, and the planner's own internal states, and to formulate goals consequent on them. Goals may be created because of a change in the world state, or in order to satisfy other goals, or to solve a problem within the planning process itself. The second of PANDORA's components is the **Plan Generator.** This proposes plans to satisfy the goals of the planner. The final component is the

Meta-theme	Situations	Meta-goals
Don't waste resources	Goal overlap	Combine plans
	Goal concord	Ally plans
	More than one plan is applicable to a known goal	Choose least costly scenario
	Goal arises repeatedly	Establish a state that fulfills the precondition of the goal as long as it keeps recurring
Achieve as many goals as possible	Goal conflict	Resolve conflict
Maximise value of goals achieved	Unresolvable goal conflict	Choose most valuable scenario
Avoid impossible goals	Circular subgoals	Resolve circularity
	Goal is too difficult	Ally with concordant goal: Resolve need

table.2.1 Themes and situations for PANDORA

Executor , which executes plan steps and detects errors. Wilensky justifies incorporating the rather novel Goal Detector by stating that it gives the planner more autonomy, and the ability to recognise internally generated goals and meta-goals. For example, if a planner could detect when it should try to combine plans to create a more efficient one, redundant actions in the final plan can be avoided.

Goal detection involves a program called the **Noticer**. This utilises a set of tests, which match specific events (ie. changes in the world, or changes within the plan). If a match occurs, then the associated theme or meta-theme is activated, and the goal most appropriate to the situation selected. Task specific constraints can be simulated by making a specific situation activate a more general principle. The Noticer detects the situation, determines which theme is violated, and then generates a preservation goal.

Having detected a goal, the system places it at the beginning of a goal queue. If there are other goals of greater urgency on the queue, a new meta-goal is created, that is, reschedule goals. This has the highest priority of all, so it is dealt with first. It causes a plan to put the previous goal in a more sensible place in the queue to be made then executed.

The Plan Generator is comprised of three parts. The **Proposer** proposes a plan that seems appropriate to the goal to be satisfied. The **Projector** simulates this plan. Since the proposed plan may not be fully detailed, or may have small errors in it, the Goal Detector is used by a **Revisor** to monitor the simulation, and to cause subplans to be generated. These subplans will make the plan more detailed, and correct any small errors. The process of producing subplans itself will involve simulation etc, so the whole thing is recursive.

The Proposer seems to have three options open to it. Normally, it will try to apply one of it's stock of standard solutions to satisfy the goal. If this doesn't work, than the normal planning mechanisms will be invoked. If these prove too weak, then the system will attempt to find a novel solution to the problem. If no solution at all can be found, then PANDORA may be able to accept a flawed plan. Goals and meta-goals can have values attached to them, so that if, say, we had an irresolvable conflict between two goals, then the system can plump for the goal having highest value, and sacrifice the other.

The main emphasis of Wilensky's work is on the theme that planning is a domain to which planning can be applied. The same type of problem solver that constructs normal plans can also plan how to construct ways to plan. What is more, the goals and meta-goals of the planner and meta-planner respectively, can be assigned values, and compared directly. More information on meta-planning and PANDORA will be found in Wilensky [WILENSKY, 1981] and Faletti [FALETTI, 1982].

2.2.14. SIPE

SIPE is intended to be an interactive planning program. It uses a hierarchical, non-linear representation (the *procedural net*) for plans. It is able to represent and reason about resources, impose constraints on variables and add to these constraints, and store alternative plans.

2.2.14.1. Plan Representation

A plan is represented in SIPE as a network of PROCESS and GOAL nodes. PROCESS nodes specify some action that must be applied to the domain model. GOAL nodes specify some fact that must be true at that point in the plan. Both PROCESS and GOAL nodes may be made more detailed by the application of an operator. Arcs in the network represent time ordering on the nodes. If several nodes are to be ordered after a PROCESS or a GOAL node, then a SPLIT node is interposed, and all of the links taken from this node. If several nodes are ordered before a PROCESS or a GOAL node, then a JOIN node is interposed, and all of the links made to impinge on this node. This network representation is rather like a hybrid of that used by NOAH and that used by NONLIN. SIPE is also able to build up a Table Of Multiple Effects (TOME), but it only does this when the resource handling heuristics are unable to resolve a conflict.

2.2.14.2. Operators

An operator is essentially a description of an incremental change to the world model. Before an operator can be applied, a set of *preconditions* must be satisfied. An operator also has *effects*. These specify facts that become true or false once the operator has been applied. There are two types of effect - the main effect, or *purpose* of the operator, and the side-effects. Side-effects may also be specified by deductive operators (see later). Lastly, an operator definition will contain a "plot" which defines the more detailed PROCESSes and GOALs that make up the operator. A simple language enables the user to specify the PROCESSes and GOALs as being in an arbitrarily complex combination of serial and parallel configurations. These more detailed PROCESSes and GOALs define the next deepest level of abstraction.

2.2.14.3. Partially Described Objects

SIPE can represent unknown objects by the constraints imposed on them. These constraints can be simple restrictions on the instantiations for the object, or they can take the form of relationships with other variables. When new operators are applied, already existing objects may have new constraints added directly, or via constraints on variables related to the object. *Evaluating* constraints is the finding of all possible object descriptions which fall within the constraints. SIPE can be called upon to evaluate constraints at any time.

2.2.14.4. Resources

If an object is specified as a resource of an operator, then one of the preconditions of that operator is that the object is not simultaneously being used by some operator on a parallel branch in the plan. One of the useful things about resources is the way in which they allow conflicts to be detected without evaluating operators. For instance, in the blocks world, we may have a problem with an initial state of three blocks, A, B and C being on the table, and a desired final state where A is on B and B is on C. One of the preconditions of the operator for putting B onto C is that B is clear. However, putting A onto B could make this false, depending on how we ordered the two actions. Non-linear planners like NOAH and NONLIN will put these actions on parallel branches of the plan, evaluate them, and then detect that a conflict has occurred. In SIPE, it is possible to specify that the block B is a resource of the action "put A on B". The resource heuristics will detect that this resource is being used on two parallel branches, without having to evaluate the two puton operators. The standard algorithm for resolving this type of conflict is to order the operator specifying the object as a resource after the operator with which it is in conflict. The user can change this ordering if so desired.

2.2.14.5. Exploring Alternatives in Parallel

SIPE has another type of node not so far mentioned; this is the CHOICE node. A CHOICE node is inserted in the net whenever choices are possible. This enables the user to specify any alternative plan, by giving the path of such choice nodes leading up to that plan.

2.2.14.6. Deductive Operators

A deductive operator has only preconditions and effects. It's main purpose is to supplement "normal" operators, by deducing the detailed effects of these operators. This saves the user the trouble of specifying the side-effects of an operator when it is defined.

2.2.14.7. User Interaction

One of the aims of SIPE is to be highly interactive. The following are some of it's user options:

(1) Determine which actions can be used to plan a given step in more detail.

(2) Test to verify that a given action can be used for further planning of a given step.

(3) Test the availability of resources for a given plan or subplan.

(4) Plan a step in greater detail.

(5) Indicate which objects to use for an action.

(6) Instruct system to select objects for an action.

(7) Instruct system to find conflicts.

(8) Rearrange plan steps to resolve a conflict.

(9) View other parts of the plan, or other plans.

(10) Change the focus to another plan.

(11) Label a plan or plan segment for future reference.

Both textual and graphical interaction are possible. If a graphics terminal is available, the system can present the user with all or part of the plan network, and can show the user what the world model looks like after the current plan has been executed.

Interaction presumably occurs after every job performed by the planner, these jobs having been created by previous interactions with the user.

2.2.15. Comments

I hope that this overview gives some flavour of the variety of planning techniques that have been used in the past.

Graph Traverser, and other state space search systems, suffer from a number of problems. Because they work incrementally, from one world state to the next, they are bound by the tyranny of detail; hierarchical techniques for simplifying the problem domain cannot be used. The evaluation heuristic must take into account all of the information about the world state, and for complex domains, the development of such heuristics could be a daunting problem.

Although MODPLAN does not have the ability to automatically refocus it's attention in the way that Graph Traverser can, it must still evaluate the the options available to it at choice points. Laying the foundations for the learning of evaluation heuristics is one of the goals of this thesis.

STRIPS breaks up the problem solving problem, by making guesses at applicable operators, and hence intermediate sets of goals. The heuristics used ~~to to this~~ were completely general. If some knowledge of the domain being planned ~~for~~ could have been incorporated, smaller search spaces, and faster planning might have been feasible. Also, like Graph Traverser, STRIPS had no way of ranking the importance of any particular goal, ie. it was non-hierarchic. Sacerdoti's ABSTRIPS, ([SACERDOTI, 1973]), was an attempt to get around this. LAWALY, of Dreusi and Siklóssy ([SIKLOSSY, 1973]), was a similar attempt.

WARPLAN, INTERPLAN and Waldinger's goal regression system were all attempts to deal with goal conflicts in a systematic way. Both WARPLAN and INTERPLAN use backtracking as their control structure, so that failure and replanning could mean repeating work already done. WARPLAN uses an "action regression" that is similar in many ways to Waldinger's idea. However, by regressing goals, Waldinger gave his system the option of selecting *any* action relevant to satisfying the regressed goal, rather than the single action that would be available in WARPLAN. WARPLAN is of especial interest, since it was one of the first planning programs to be written in PROLOG, the implementation language of MODPLAN.

One of the major problems with PLANNER is the backtracking regime under which it runs. If no traps for failure messages have been planted, many wasteful attempts may be made to generate a plan before the decision point responsible for the failure is found. And when it is found, there is no information about the cause for the failure available, so the system will blindly try the next alternative. If traps are to eliminate these problems, they must be of sufficient number and variety to cover all possibilities of failure. With either straight backtracking or under failure messages, all planning between the failure point and the decision point must be discarded, whether it is relevant to the failure or not.

NOAH avoided the assumptions of linearity made in previous planners, simplifying the detection of conflicts. NOAH also made use of constructive critics, which look at a partial plan and suggest improvements which could be made to it. However, there are several limitations on the types of conflicts which can be resolved by these critics. NOAH has no search space; once choices have been made, they cannot be undone as a result of failure in the planning process.

NONLIN overcomes some of the problems encountered in NOAH. It recognises all possible conflicts, and has at it's disposal a complete set of conflict resolving algorithms. It does have a simple extended backtracking control structure, so that errors made during planning are not fatal, and alternative plans can be produced. However, in many circumstances, backtracking is not a very satisfactory response to failure, and a more intelligent control structure is desirable. NONLIN's Task Formalism affords a considerable improvement in readability over NOAH's SOUP code for operato definitions. NONLIN is the planner "modelled" by MODPLAN.

The decision graph, as used by Daniel, offers a more versatile contro structure than backtracking when used to deal with plan-time failures. How ever, deciding which choice was responsible for a failure is a problem, as it i with PLANNER's failure trapping. The most interesting thing about these systems, as far as MODPLAN is concerned, is that the types of choices that a planner makes must be made explicit. In neither of the two systems reviewed, however, are the complete set of choice types considered. This may have been due to the fact that in the domains with which the authors were working, only a limited range of choice types were required.

NASL does not store information on the choices it has made, and hence backtracking etc. on failure are not possible. However, to compensate for this, it is able to set itself the task of rectifying a failure, treating this as another sub-problem to solve. Like MODPLAN, NASL uses policies. However, it treats them as secondary tasks, on the same level as tasks for jobs to be done in the domain. In MODPLAN, policies are regarded as part of a set of meta-level goals, and are not mixed with domain-level goals. There are a number of ways in which policies can affect problem solving in NASL, one of which is to change the way in which choices are made. In MODPLAN, this is the *only* way in which policies can affect problem solving.

MOLGEN and PANDORA are two planners that were designed not only with planning in mind, but also with an eye on the problem of *how* to plan. Both of them used some kind of meta-planning. Wilensky's "meta- themes" are similar to the policies used in MODPLAN, although MODPLAN's policies are active all of the time, and cannot be switched about to suit a particular situation.

The feature of MOLGEN that most strongly comes under fire from Wilensky is that, for each level in the control structure, a different (and, presumably, increasingly specialised as we ascend the structure) problem solving system is used. It is rather paradoxical that one of the quotes in Stefik's paper ([STEFIK, 1981b], p 146) recommends that the *same* problem solving language is used at each level.

SIPE is a non-linear planner, similar in many ways to NOAH and NONLIN, although it's ability to detect and correct goal conflicts is poorer than NONLIN's. It's real strength lies in the interactive facilities provided, which give the user considerable control not only over how choices are made, but also over what order they are made in. Some of SIPE's interactive facilities were felt worth emulating in MODPLAN. SIPE also gives the user a means of exploring alternative plans, with it's CHOICE nodes. Although undoubtedly a useful feature, this does mix plan level information with meta-level information. It would have been better to have kept a separate tree of choices.

These systems have tended to be applied to fairly simple domains - such as the block stacking and box pushing ones used as examples in this discussion. However, some practical applications of problem solving systems have been made:

(1) Tate's NONLIN (discussed earlier in this chapter) was used for planning turbine overhauls ([DANIEL, 1977, DANIEL, 1982]).

(2) The DEVISER system of Vere ([VERE, 1981]) plans missions for deep space probes.

(3) McDermott ([MCDERMOTT, 1978]) wrote a program called NASL (which is reviewed earlier in this chapter) for designing electronic circuits.

(4) Hearsay II was written by Erman et al ([ERMAN, 1980]) for the understanding of human speech.

(5) Stefik's MOLGEN ([STEFIK, 1981a, b]) was used to plan experiments in molecular genetics. The control structure of this program is described earlier in this chapter.

(6) Nii and Feigenbaum wrote SU-P ([NII, 1978]) for analysing the results of X-ray crystallography data in order to determine protein structure.

2.3. Review of Selected Learning Programs

2.3.1. Introduction

It will be useful to look at some existing learning programs, to see how applicable they would be to generating heuristics, and how ideas contained within them have influenced MODPLAN's development. For the programs that learn by example, the classification scheme used by Cohen and Feigenbaum in their "Handbook of Artificial Intelligence" Vol 3 ([COHEN, 1982]) will be employed. This lists three different classes of program that learn rules from examples:

(1) Learning single concepts.

(2) Learning multiple concepts.

(3) Learning to perform multiple step tasks.

Cohen and Feigenbaum further subdivide these classes into learning programs that are data-driven, and learning programs that are model-driven.

Of the data-driven programs in class (1), we could list Mitchell's version space, and the closely related techniques of Winston ([WINSTON, 1970]), Hayes-Roth ([HAYES-ROTH, 1977]) and Vere ([VERE, 1975]). Also in this group are the CLS/ID3 family of inductive learning programs, developed by

Hunt and Quinlan. Of the model driven programs, mention could be made of Dietterich and Michalski's INDUCE 1.2.

In class (2), we shall look at Michalski's AQ11 and Lenat' AM.

In class (3), we shall look at LEX of Mitchell et al. Other important work in this area includes Langley's SAGE ([LANGLEY, 1982]), and Waterman's poker player ([WATERMAN, 1970]), but reviews of these programs have been left out for the sake of brevity.

Many other programs that are able to learn from examples have been written; a useful review of some of these can be found in Bundy and Silver ([BUNDY, 1982]).

Finally, a review of Mostow's operationalisation program, FOO ([MOSTOW, 1979]), is presented. This program exhibits a simple form of learning ability. It is relevant because some of the concepts and terminology used to describe MODPLAN's choice making element are derived from it.

2.3.2. Version Spaces (and related techniques)

The version space approach, as propounded by Mitchell, works roughly as follows. The learning program has at it's disposal the space of all possible rules. Rules, and the examples used to train the system, use the same (predicate calculus) representation. The rule space is partitioned by two sets; the set G of most general rules, and the set S of most specific rules. Notionally, these sets define two "lines" through the rule space. The portion of the rule space *between* these lines represents the currently acceptable plausible rule set, and is called the **version space**. The portions of the rule space above and below these lines contain rules which have been rejected by the learning algorithm.

Initially, G will simply be "true" and S will contain the currently available set of training instances, ie., G will be at it's most general, and S at it's most specific. The version space will contain the whole of the rule space. When supplied with a positive training instance, the system will attempt to find rules more general than the current set of training instances. The minimum possible generalisation is done, guided by a generalisation hierarchy, on the set S of most specific rules. Negative training instances cause the system to do the converse; the system will attempt to specialise the set of most general rules (G). Repeated positive and negative training instances gradually force S and G together. When S and G become equal, the learning process is complete. It is assumed that S and G converge onto a set containing a ***single*** rule that explains ***all*** of the examples.

Refer to [MITCHELL, 1977] for more information.

2.3.3. Inductive Learning

The CLS/ID3 family of learning programs ([HUNT, 1966, QUINLAN, 1983]) represent their rules as a decision tree. Each node in the tree corresponds to an attribute; arcs correspond to the values that attributes can take. Leaf nodes are the classifications, either "yes" or "no". Such a rule can be used for classifying an instance. The system will first compare the attribute at the root node of the decision tree with the corresponding attribute in the instance. The value of this attribute will lead the system onto the appropriate next node in the decision tree, which will be used to check another of the instance's attributes. Eventually, all of the instance's attributes will have been checked in this way, and the classification at one of the leaf nodes assigned to it.

The decision tree initially contains only a root node. When learning, the system must be presented with the complete training set, C. Using a heuristic, one of the attributes is chosen to split C into subsets, one subset for each possible value of the attribute. For each of these values, a node is put into the decision tree. The process of choosing an attribute, splitting, and adding nodes to the decision tree is then performed for each of these subsets, and successively repeated, until single examples, with their classifications are found. These classifications are placed in the leaf nodes of the decision tree; the learning process is now complete.

2.3.4. INDUCE 1.2

INDUCE 1.2 ([DIETTERICH, 1981]) is a model-driven concept learner. Training instances are transformed, so that they are in the same representation as the rules. Given a (complete) set of training instances, INDUCE 1.2 selects a random subset of H instances. It then performs a "beam search" upwards in the rule space; this works by repeated generalisation and pruning of rules. Every possible generalisation of each instance in H is found, by removing conditions from rules. Then, any rules that result from this that are unacceptable, are pruned, using a model dependent heuristic. After pruning, the remaining, more general rules are tested, to see if any explain all of the examples. Any which do are stored in a set C. The process of generalising and pruning is now repeated for the more general rule set, to produce a more general set still, and so on, until C has attained some preset size, or no more generalisation is possible.

2.3.5. AQ 11

AQ 11 ([MICHALSKI, 1978]) is a program that is capable of learning *multiple* concepts. AQ 11 applies a single concept learning algorithm for eac! concept that it is aware of; this algorithm is similar to the version spac technique. It classifies all examples leading to a desired concept as positiv training instances, and all other examples as negative training instances. A(11 will build up a set of concepts, each of which is the most general possible. This means that the concepts may overlap - a given instance may cause two or more concepts to fire. If desired, the user can make AQ 11 build up non-overlapping concepts. During the learning of a given concept, both instances not leading to the desired concept *and* instances covered by concepts already learned are taken to be negative. Thus, the areas of positive instances covered by the concepts learned will never overlap. However, this does mean that the first concept learned will cover a large number of instances, the second concept rather fewer, the third concept even less, etc.

2.3.6. AM

Like AQ 11, AM is a program capable of learning multiple concepts. It is a complex program, designed to discover "interesting" concepts in mathematics. It is initially provided with about 100 set theory concepts, and over 200 heuristics for guiding it's search. It learns iteratively, generating it's own examples. It is capable of generalising or specialising concepts, and creating new concepts. An outline of it's operation is as follows. Each concept in AM is represented as a *frame*. This is rather richer than the usual condition/action representation for concepts. It contains a number of *slots* , including a DEFINITION slot, which can be used to test to see if the concept is applicable, GENERALISATION and SPECIALISATION slots, that point to more general and more specialised concepts respectively, a CONJECTURE slot, which gives conjectural links with other concepts, and a WORTH slot, that contains a numerical estimate of how "interesting" this concept is. AM has an agenda of tasks to be performed, which are all of the form "fill or modify a slot of concept C". These tasks have a numerical "interestingness" rating associated with them, and AM selects the most interesting to execute first. The heuristics that fill or modify slots are also capable of adding new concepts, and putting new tasks onto the agenda. The learning cycle of AM car be summarised thus. A set of examples is generated to test a concept. For each example, depending on it's applicability, the system can decide to change the interestingness of the concept, create new concepts, or create new conjectures. Any knowledge thus gained can be used to update other concepts in the system.

Refer to [LENAT, 1976] for more information on AM.

2.3.7. LEX

LEX ([MITCHELL, 1981]) is the only multiple concept learning program that we will be looking at in detail. As an entire system, it is very close in spirit to what I wish to provide for MODPLAN. LEX is a problem solving program that is able to improve it's performance in the solution of simple integration tasks. It works thus. An integral to be solved is presented to the system. Using a uniform cost algorithm, plus any heuristics learned from previous examples, it attempts to construct a minimum cost path through the solution space, by successive application of integral simplification operators. These operators are given to the system initially, and include the basic rules of algebraic simplification as well. If a path is found, the system goes into a learning phase. At each choice point along the minimum cost path, all possible decisions are taken. Any which do not result in a path less than 1.15 times the length of the minimum cost path (a length chosen, probably empirically, by Mitchell) are treated as negative training instances. All others, including the one on the minimum cost path, are taken to be positive training instances. A version space-type learning algorithm uses these training instances to refine it's G set and generalise it's S set. Once every choice point on the minimum cost path has been treated in this way, the system is ready to accept a new example, from either the user or an example generating program.

2.3.8. Mostow's Work on Operationalisation

As part of a project to implement a program that is able to take advice, Mostow ([MOSTOW, 1979]) wrote a program called FOO (First Operational Operationalisationer).

Using advice is seen by Mostow as a five stage process. It is entered in natural language form and **parsed** into a syntax tree. This is then **interpreted** into an internal representation (the system's concepts). The interpreted advice is **operationalised**, so that it can be applied to produce recommendations on the decision at hand. Different pieces of advice can then be **integrated** by combining these recommendations, and then **applied** to guide the system's decision making.

FOO was aimed at providing techniques for operationalising advice for the card game "hearts". The problems of tackling parsing, interpretation, integration and application are not considered by FOO. It must be presented with advice that is pre-interpreted, and it is unable to do anything with the advice once it has been operationalised. Mostow identifies two classes of operationalisation - static and dynamic. Operationalisation performed statically does not depend on the state of the game, whilst dynamic operationalisation exploits information obtained from the current game.

FOO uses a uniform LISP-like representation for it's knowledge. The representation uses as it's basis a set of **meta-functions**. These are domain independent concepts, such as:

(EXISTS x S P)	=	there exists an x in the set S such that P is true.
(ALL e)	=	Find the set of all possible values for the expression e.
(PR P)	=	find the probability that P is true.

A set of **domain concepts** is defined in terms of these meta-functions. For instance, in the domain of "hearts", the definition of the concept of "points" is:

```
(DE POINTS (CARD)
       (CASE (( = CARD QS) 13)
           ((= (SUIT-OF CARD) H) 1)
           ((= CARD JD) -10)
           0)
```

This can be interpreted as meaning "the point value of a card is 13 if it is the queen of spades, 1 if it is in the suit of hearts, -10 if it is the jack of diamonds and zero if it is anything else".

The same representation is used for the **constraints** on what the system can do (the game rules), and the **heuristics** used to guide play. As an example,

```
(DE AVOID-TAKING-POINTS (P)
       (AVOID P (TAKE-POINTS P)))
```

is a heuristic that will tend to make the system avoid taking point cards.

Advice is also expressed in the same way as concepts. The purpose of operationalisation is to take a piece of advice which is non-effective, and to transform it so that it becomes executable at the domain level. Advice can be non-effective for many reasons. For instance, it may depend on non-observable information, such as the cards possessed by an opponent, or it may be too complex to evaluate precisely, yet still amenable to approximation.

As well as a means of representing concepts, FOO also has a formalism for operationalisation methods. These methods are drawn from the inference techniques that are used in AI, and FOO has about 200 such methods available to it. Each method can be regarded as a way of reformulating an expression, in order to create sub-expressions that can in turn be reformulated etc., until an operational form is arrived at.

One of the problems in operationalising advice is selecting an appropriate method for a given expression, and mapping the variables in the expression onto those of the method. In FOO, this is performed manually.

FOO has five classes of operationalisation method:

(1) **Goal specific operationalisation rules.** The expert can phrase the advice given in operational terms. For instance, the advice "to avoid taking points, play a low card" in hearts gives an operational method, "play a low card", that can be applied directly in decision making.

Systematic evaluation. The expression representing the advice is expanded by substituting the expansions of known concepts for the sub-expressions of the main expression. For example, if the system has a concept of avoidance (defined in terms of finding an action that has a small probability of leading to a given event), then the following is a possible evaluation for "avoid taking points":

```
(AVOID ME (TAKE-POINTS ME))
evaluates to
(SOME ACT (ACTIONS-OF ME)
      (SMALL (PR-LEAD-TO ACT
                   (TAKE-POINTS ME))))
```

(3) **General operationalisation methods.** These methods find a method of operationalising an expression that is more general than the expression being considered, and then applying the general method to the specific expression. For instance, if we wish to "avoid taking points", we can find some technique for avoiding things in general. One such technique is to find the conditions that cause an event, and then prevent them. This is known as fault tree analysis.

(4) **Weak methods.** One weak operationalisation method discussed by Mostow [MOSTOW, 1979] is the heuristic search method. It can be used in situations such as finding out if a given piece of information will become true in a future game state. In this instance, it can be used to generate a sequence of plausible game states, terminating with the state in which the information is true.

Another weak method is approximation. As an example, we may wish to make the probability that a given action will lead to the player taking points small. This could be expressed as

```
(SMALL (PR-LEAD-TO ACT (TAKE-POINTS ME)))
```

If we have difficulty in operationalising this, we can make the approximation:

```
(NOT (LEAD-TO ACT (TAKE-POINTS ME)))
```

This is effectively replacing a small probability with a zero probability.

(5) **Dynamic operationalisation.** Operationalising advice whilst a game is being played can be considerably easier than producing general advice statically. This is because knowledge from the game constrains the number of possibilities that might have to be considered during operationalisation. For instance, since the machine knows what cards it has in it's own hand, the possible cards that it can play are constrained to be within this set.

2.3.9. Comments

The following comments give some idea of the advantages and disadvantages of the various learning programs in general terms. Indications are also given about how useful the programs would be if applied to the problem of learning choice making heuristics in planning.

The disadvantages of the version space technique are:

(1) It requires that examples and rules use the same representation.

(2) It is very sensitive to noise (ie. inaccurate examples).

(3) It cannot learn disjunctive concepts.

Mitchell has proposed extensions of the basic technique to alleviate (2) and (3). (1) could probably be circumvented if a uniform method for converting examples into (say) predicate calculus form were available.

However, this technique is a promising candidate for the learning of heuristics, since the learned rule can be improved incrementally in the light of new examples.

The nice thing about CLS/ID3-type decision trees is the way in which they enable disjunctive concepts to be represented naturally. This is a feature that may well be useful in representing choice making heuristics.

One of the major disadvantages of this learning method, however, is that all examples must be presented at once; if it were desired to learn incrementally, then it would be necessary to store all previous examples, and then perform an induction over *all* examples if any new examples were encountered.

The INDUCE 1.2 technique has the advantages of being quicker than the full version space method, and using less memory. It also has a good noise immunity. Since the examples obtained from a trace of a planning run will often contain "untried" decisions, to which it is not possible to assign definite "success" or "failure" tags, this noise immunity would be useful. However, it does have the disadvantage that all training instances must be presented at once, so to use it in a planning domain, where training occurs at unpredictable intervals, storing of all previous examples would be necessary.

The problems with AQ 11 are that it requires all examples to be presented at once, and that, for problems with a large number of attributes, it is significantly slower than the CLS family of learning programs.

In it's unadulterated form, AM could not be used in the planning domain, since it does not allow the user to present examples. Furthermore, even if we substituted "find optimal plans" for the "interestingness" criterion, a large body of heuristics would have to be written to guide the learning system. Since we may well want the system to learn heuristics that are only applicable for planning in a given domain, we might have to write many heuristics for the learning system that are dependent on the domain... which rather defeats the purpose of a system that is supposed to learn it's own heuristics! Another problem with AM generally is that the heuristics supplied are intended to guide the search when concepts are roughly at the same level as the initial set of concepts. When the system starts to develop higher-level concepts, these heuristics become rather ineffectual. Lenat's

EURISKO project is an attempt to rectify this, by turning the heuristics themselves into concepts, and thus giving the system the ability to discover more powerful heuristics (see [LENAT, 1980]).

Some changes would have to be made to apply the ideas in LEX to a planning domain. First, different choice points may represent different ***types*** of choice, and hence a different learning process will have to take place for each type. Second, there may be no obvious way of generating a least cost path; the user may well be happy as long as *some* path is found. And, combining this with the fact that the solution space could be very large, we might not wish to use exhaustive search techniques for identifying negative training instances; it may be better simply to *ignore* untried decisions.

Because FOO has a large number of domain independent operationalisation methods, it is, in theory, applicable to many domains. Supplying FO with domain concepts, constraints and heuristics for a new domain wo probably be quite a big job. This is a problem shared by other progra requiring large amounts of domain knowledge. The lack of a parser interpreter for acquiring advice is probably not an overwhelming disadvantage. "Policies" in MODPLAN are very similar to the "advice" used by FOO. The concept of "operationalisation" has been adopted by MODPLAN to describe the way in which a given policy will affect choice making for a given choice type.

CHAPTER 3

The Operation of MODPLAN, and the Choices That it Makes

3.1. Overview

One of the axioms behind the work presented here is that during planning, choices of various types are made. For instance, a NONLIN type planner can be called on to choose between a number of alternative methods of correcting an interaction; thus, we have identified the existence of a "choose interaction correction method" choice type. The same program may also, on a different occasion, have to choose which of a number of eligible plan nodes to expand; we have now discovered a second choice type, "choose a plan node to expand". A choice type analysis along these lines was made for a non-hierarchical version of Tate's NONLIN. MODPLAN "modelled" NONLIN in terms of the choice types thus discovered.

MODPLAN's operation is controlled at the highest level by a recursive function, which performs one recursion per choice type. At each level in the recursion, a list of **decisions** is generated. The process of making a **choice** consists of selecting one of the decisions, and executing any action associated with it. These actions cause changes to be made incrementally to the structures representing the plan A simple scheduling function decides on the *type* of the choice at a given level of the recursion, and also detects plan completion.

This chapter tackles the following issues:

(1) An analysis of the types of choice made by NONLIN during plan generation.

(2) How information on choices made is stored in MODPLAN.

(3) MODPLAN's operation, including it's control structure, the way in which information is passed from one choice to another and the workings of the Plan Modification Formalism.

3.2. Choices Used by MODPLAN

First of all, then, what choices are made during the planning process? An analysis of a non-hierarchical version of NONLIN revealed twelve different types of choice, including 'start', a dummy initial choice, 'stop', a choice generated when a plan has been found, plus others, eg. choose interaction correction method, select a node to expand.

The following table gives, for each choice type identified, a "nickname", which will be used throughout this text, plus a description of what happens at such choices.

Choice Type	Description
'start'	Dummy initial choice. This is used mainly for convenience, to make the control structure a bit simpler.
'choosexp'	Choose a plan network node to expand. The system examines the current list of expandable nodes, and selects one for expansion.
'choosexm'	Choose expansion method. There are three possible methods of expanding a network node ie. "allow the fact at a node to be already satisfied", "satisfy the fact at a node by making links", and "use an operator". The system will select one of these, and use it to expand the node chosen by the last 'choosexp' choice.
'chooseoper'	Choose an operator. If it was decided that a node should be expanded by an operator, then the system will select an operator from the list of operators specified by the Task Formalism.
'choosecontrib'	Choose a contributor to a fact. A fact may not be definitely true at a node, but it may be possible to find potential contributors, and then to correct any interactions that result from choosing one of these. It may also be necessary to link from the potential contributor to the node at which the fact is to be satisfied.
'choosefact'	Choose a fact to instantiate. All of the expansion methods will generate conditions to be satisfied, in the form of facts which must be true at the node expanded. These facts, which make up the conditions on an expanded node, may not be fully instantiated; it is the function of the 'choosefact' choice to select one of the facts, so that instantiations can be made for it.
'chooseinst'	Choose instantiations for a fact. The question answering system (see the description of NONLIN in chapter 2) is used to find all nodes at which effects arise which would satisfy the fact immediately. Since the fact may not be fully instantiated, there may be several different effects that would satisfy it; it is the purpose of 'chooseinst' to select one of these, and hence select the instantiations for the fact.

Choice Type	
'chooseint'	Choose an interaction to correct. If one of the plan node expansion methods has caused interactions to arise within the network, the system's interaction correction mechanisms need to be called to action. Given a list of interactions outstanding, 'chooseint' will select one of these to be corrected.
'choosecorr'	Choose an interaction correction method There are seven possible ways in which MOD PLAN can correct interactions (these are th same as the methods used by NONLIN; se chapter 2), and one of these must be selecte to correct the current interaction.
'choosecond'	Choose an 'unsupervised' condition to satisfy. Operator definitions may specify that certain conditions are of type 'unsupervised'; in this case, the system makes no attempt to satisfy them when the operator is used to expand a node. Instead, they are put into a list, and satisfied when all other planning is complete. 'choosecond' will select one of these conditions from this list.
'choosesatis'	Choose a method to satisfy an 'unsupervised' condition. There are two methods available for satisfying 'unsupervised' conditions; "allow condition to be satisfied in current plan", and "satisfy condition by making links". 'choosesatis' will select one of these for the condition currently under consideration.
'stop'	Stop planning on this branch in the tree. If a plan is successfully generated, then the current choice becomes 'stop'. No further planning can be done from this point, but the user can refocus the attention of the planner, and generate alternative plans.

These choice types are probably applicable to a much wider set of planners than just NONLIN, although the way in which they are ordered (the control structure) may be different.

3.3. Choice-Point Tree

The choices made during MODPLAN's operation are stored in a tree. Nodes in this tree correspond to choice points, and are represented as PROLOG structures. Arcs correspond to a chronological ordering on the choices made, and are represented by pointers in the choice-point structures. At each choice-point, six pieces of information are maintained:

? What is a Prolog structure?

(1) A unique **number**, which identifies the choice-point as a node in the tree.

(2) The **type** of the choice dealt with at this point.

(3) A pointer to the **parent** choice-point tree node.

(4) A list of all possible **decisions** that could be made at this point.

(5) The **time** at which the choice was made.

(6) A **label**, which may be set to an arbitrary string by the user, if so desired.

The list of decisions (in (4) above) is very important, and it performs two functions. Firstly, it specifies all possible selections that could be made at a choice point. Second, for those selections which have been tried, it gives an indication of whether they failed, and, for those that didn't, what the numbers of the subsequent (child) choice-points are.

Each decision in the list is a structure, containing the following arguments:

(1) **Status.** This tells the system the status of a decision. If it is uninstantiated, then the decision has not yet been tried. If it has been instantiated to 'fail', then the system knows that the decision has been tried out, and has failed. If it is instantiated to 'child(N)', then the decision has been tried, succeeded, and leads onto choice-point number 'N'.

(2) **PMF.** This stands for "Plan Modification Formalism". It forms the "action" part of the decision, and specifies the effects that executing this particular decision will have on the plan.

(3) **Matchpattern.** This is a structure used to pass arguments to the PMF.

(4) **Merit.** This is a numerical figure of merit assigned to the decision during choice making. Within a list of decisions, the one with the highest figure of merit is selected for execution first of all by the automatic choice making function.

3.4. MODPLAN's Operation

3.4.1. Introduction

The previous sections described the *choice types* used by MODPLAN, plus the *meta-level information* it stores (ie. the choice-point tree). In this section, we move on to MODPLAN's **operation**.

MODPLAN's operation is centred around the making of choices. The core of the system is a recursive function, called 'plan', which when called, is given the current choice type, selects a decision, executes it, and then calls 'plan' again. Recursion ceases when the current choice type is found to be 'stop', at which point it is assumed that the system has arrived at a valid plan.

3.4.1.1. MODPLAN's Control Structure

Before a choice is made, a function, which forms part of MODPLAN's control structure, will have been called to decide on the *type* of this choice. The algorithm currently used to do this is not very sophisticated, since, in the majority of cases, the type of the next choice is dependent on little more than the type of the choice preceding it.

The way in which the system orders choice making is roughly as follows:

(1) Find a node in the plan network to expand; if none, goto (6)

(2) Select a method to expand this node

(3) Find instantiations for variables within the expansion

(4) Check for and correct any interactions introduced

(5) Goto (1)

(6) Satisfy any 'unsupervised' conditions.

(7) Stop.

Each step represents a choice to be made. If failure occurs, the system backtracks to the previous step (choice-point). This ***control structure*** is based on a non-hierarchic version of Tate's NONLIN. Steps (1) to (5) form the basic cycle of the control structure. The user only has a say in the ***way*** in which these choices are made; the *ordering* of choice types is embedded in the program code.

3.4.1.2. Making Choices

Once the *type* of a choice has been decided, we can set to the task of *making* it. MODPLAN uses what is essentially a generate and test algorithm to do this. Given the type of the choice, plus information from other choices and, perhaps, the plan, a set of possible *decisions* is generated. The user is then presented with a list of options, which will usually lead to the selecting of a decision. Having got a decision the system will then attempt to execute it. This may mean doing nothing, but a change to the plan is often entailed.

If this decision cannot be executed, then the system must decide what to do; this could involve choosing one of the other decisions, or failing, and back tracking to a previous choice.

Making choices, trying them out, and deciding what to do if they don't work, are all quite closely linked. At each choice-point, the system goes through the following sequence:

(1) Remove from 'Cinfo' any information from previous choices that is pertinent to the current one.

(2) Generate a list of decisions to try.

(3) Test the decision list, using a failure detection mechanism. If the list does not pass this test, then the system will backtrack to the previous choice. Otherwise,

(4) Create a node in the choice-point tree, containing a copy of the decision list and a pointer to the previous choice (the **parent** choice). If any instantiations are made to the original decision list in subsequent parts of the choice making process, these will be transmitted to the decision list at the choice-point, since these lists share the same variables.

(5) Select one of the decisions (ie. make a choice).

(6) Try the decision out. This means taking the piece of Plan Modification Formalism (PMF) associated with the decision, and attempting to make the changes to the plan specified by this PMF (ie. executing the PMF).

(7) If the PMF executes successfully, then go to (8). If it proves impossible to execute the PMF, the status of the decision, previously uninstantiated, will be instantiated to 'fail', and the system returns to (3).

(8) Any information that might be useful to future choices is put into a structure called 'Cinfo'.

(9) The type of the next choice is determined.

(10) Choice making is exited successfully.

The issues tackled in the following sections correspond quite closely to the stages in the above sequence, although the order may be a bit different.

3.4.2. Deciding on the Type of the Next Choice

The type of the *next* choice is established at the *current* choice point. The function for doing this takes as input the current plan, the current choice-point tree, and the list of information passed down from previous choices (called 'Cinfo'). It outputs the type of the next choice to be made. The next choice type is dependent largely on the types of recently made previous choices. Other factors that may influence it are the number of outstanding expandable nodes, the number of outstanding uninstantiated facts, the number of outstanding uncorrected interactions and the number of remaining unsatisfied 'unsupervised' conditions.

The following table gives, for each choice type, algorithms for finding the type of the next choice.

Current choice type	Algorithm for finding next choice type
'start'	If there are any expandable nodes in the plan network, then 'choosexp', else if there are any remaining unsatisfied 'unsupervised' conditions, 'choosecond, else 'stop'.
'choosexp'	'choosexm'
'choosexm'	If the expansion method chosen was "allow fact at a plan node to be already true", then 'start', else if the method was "make links to satisfy a fact", then 'choosecontrib', else 'chooseoper'.
'chooseoper'	'choosefact'

Current choice type	Algorithm for finding next choice type
'choosecontrib'	If last decision was 'choosexm', then 'choosefact', else if last decision was 'choosesatis', then 'start'.
'choosefact'	If the list of facts for which instantiations are to be made is not null, then 'chooseinst'. Otherwise, if the current choice is directly or indirectly dependent on either a 'chooseoper' or 'choosecontrib' choice type, then 'chooseint', else 'start'.
'chooseinst'	'choosefact'
'chooseint'	If there are no more interactions to correct, then 'start', else 'choosecorr'
'choosecorr'	'chooseint'
'choosecond'	'choosatis'
'choosesatis'	If the satisfaction method chosen was "satisfy the condition in the current plan", then 'start', else 'choosecontrib'.
'stop'	'stop'

This is what was done but why?

These "next choice type" finding algorithms, plus the failure detection and propagation mechanisms to be described later on, form MODPLAN's control structure.

3.4.3. Plan Modification Formalism

3.4.3.1. Introduction

The planner has at it's disposal a *Plan Modification Formalism* (PMF), which can be used to specify changes to be made to the plan. It is, in effect, a simple programming language, with about a dozen primitive plan altering functions, the same number of high-level plan altering functions, and six conditionals, that allow it to test for certain conditions relating to the state of the plan.

A piece of PMF is a PROLOG structure, with five arguments:

(1) **Name.** A string, used for identifying the purpose of the PMF.

(2) **Pattern.** A structure, used for passing arguments to the PMF.

(3) **Ident.** A string, that identifies the means by which the purpose of the PMF is achieved.

(4) **Body.** A list, containing lower-level PMF to be executed in order to achieve the purpose of this piece of PMF. The list is null for primitive PMF.

(5) **Applics.** A list of the applicability conditions that must hold before this piece of PMF can be executed.

Primitive PMF is defined directly in terms of the effect it will have on the plan. Examples are the PMF that adds a new node to the network, and the PMF that removes contributors to conditions in 'Gost'. **High-level PMF** is defined in terms of the lower-level PMF that would have to be executed in order to achieve it. This lower-level PMF may be primitive, or it may contain further pieces of high-level PMF. This allows PMF definitions to be nested. Examples of high-level PMF are the PMF that implements the expansion of a node specified by an operator, and the PMF that breaks ranges in the plan.

PMF is intimately linked with the "decisions" that arise at choice-points. The making of a choice involves selecting one decision (from a list of possible decisions) for execution. A decision has associated with it an action part, which describes what will happen to the plan if the decision is executed; this "action part" is a piece of high-level PMF. "High-level" PMF, stored in the PROLOG database, is also used as a basis for generating the lists of decisions at choice-points. There may be several pieces of PMF defined with the same name. Each piece of PMF with a given name represents a different action that could achieve the same purpose. When decision lists are generated, every piece of PMF with the appropriate name will be considered as a candidate for the action part of a decision. The 'Applics' test associated with a given piece of PMF will be applied, and if it succeeds, the PMF will be accepted.

Normally, high-level PMF is specified by the user, who will, presumably, be reasonably conversant with PROLOG. However, the PMF associated with 'chooseoper' choices is generated by the system. Because it is a more natural representation than PMF, the NONLIN Task Formalism has been preserved for writing operator definitions. It is translated into PMF before being used by the planner.

Executing PMF is a fairly simple process. The 'execute' function takes as input the current plan, and the PMF to be executed. As output, it produces the plan as modified by the PMF. Execution is recursive. The system first checks to see if the PMF is primitive. If it is, then the appropriate change is made to the plan, and 'execute' is exited straight away. If the PMF is high-level, then the system performs an 'execute' on each of the pieces of PMF in the high-level PMF's body, which results in the plan being incrementally changed. The PMF in the body need not be primitive - it can be high level, too - hence the recursive nature of 'execute'. If, at any stage of execution, it should prove impossible to execute one of the pieces of PMF, the system will backtrack, and try alternative ways of executing previously tried pieces of PMF. Thus, the 'execute' function is able to behave as a simple PROLOG-like interpreter.

3.4.3.2. Primitive PMF Operators

There are currently 13 primitive PMF operators. They are defined within an "execute" function, and are not accessible to the planner for direct examination. Primitive PMF is dependent mainly on the plan representation. For the current version of MODPLAN this representation is the same as in NONLIN, ie three structures, the plan network, the table of effects and the goal structure. The kinds of operation that primitive PMF will be called on to perform will be to add, remove, and change entries to these three structures. The names of these operators, along with the effects that they have on the plan, are tabulated below.

PMF name	Effects on plan
'makenode'	This function adds a new node to the plan network. The network is represented as a list of node structures. Nodes are numbered consecutively; reordering and removal are never allowed. So to add a new node, we instantiate it's number to be that of the first node in the list plus one, and then prepend it onto the list.
'replacenode'	In replacing a node, the aim is to update the type, pattern and value, leaving it's number, predecessor node list and successor node list unchanged.
'makelinks'	To make links to a node, there are two tasks to be performed - add the new links to it's predecessor list , and add it's number to the successor lists of all the nodes from which links emanate.
'remakelinks'	In remaking links, we want to transfer all links currently impinging on a node 'N' to a node 'N1', leaving 'N' without any predecessor nodes.
'changetype'	Changing the type of the node is a fairly simple operation. It is used when 'goal' nodes are changed to 'phantom' and vice-versa.

PMF name	Effects on plan
'makeeffect'	To put a new effect in 'Tome', we must first check to see that there is not an identically instantiated entry for the same node already in existence. As long as there is no such entry, the effect is added onto 'Tome', otherwise, nothing is done.
'tometidy'	The 'tometidy' PMF will, given a plan node number, tidy up the effects for that node. This is mainly for use at the instantiation of variable stage, since it is possible that, when variables are instantiated, a newly inserted effect may become literally identical to existing effects for the same node. 'tometidy' will compress all such identically instantiated entries to one single effect.
'makecond'	Inserting a new condition into 'Gost' is a fairly involved business, since 'Gost' has quite a complex structure, and it is necessary to check if a condition already exists at the current plan node with the same pattern and value as the new condition. If such a condition **is** found, and it's type is preferred to that of the new condition, then 'Gost' is left unchanged. If such a condition is found, but it's type is **not** preferred to the new condition, then the existing condition is removed, and the new condition added into 'Gost' at the appropriate place. If there is no already existing condition, then the new condition is simply added into 'Gost'.
'remcond'	A condition is removed from 'Gost'. This is only done for 'phantom' conditions.
'addcontribs'	New contributors are added to an existing condition. Used once the instantiations for a fact have been decided.
'delcontribs'	Contributors are removed from an existing condition. Used in interaction correction.
'changecondnum'	The node at which a condition applies is changed. This is usually associated with the insertion of new nodes into the plan net; the first of the inserted nodes inherits the conditions on the node being replaced by the new nodes.

3.4.3.3. High-Level PMF Operators

The form of high-level PMF is dependent mainly on the the types of choice that are defined for the planner being modelled. It is used in generating decision lists and in executing the decisions selected at choice-points.

Fourteen pieces of high-level PMF have been defined so far. The first is a "dummy", that does nothing at all. Twelve correspond to NONLIN's choice types. The last, 'breakrange', is used during interaction correction. All of the high-level PMF is declaratively defined as assertions in PROLOG's database, and the planner is able to examine it during decision generation. The following table gives the names of the PMF, plus short descriptions of their applicability conditions, effects and purposes.

PMF name	ffects on plan	Applicability tests
'dummy'	This PMF has a null body; it has no effects on the plan. It is used in decision generation, to ensure the continuity of the choice-point tree for those choices that generate no decisions.	No tests required.
'start'	A single decision is generated at 'start' choice-points, containing a piece of 'start' PMF. It has no effect on the plan.	No tests required.
'expandable'	The 'expandable' PMF is associated with the decisions at a 'choosexp' choice point. It has no effect on the plan.	No tests required.

PMF name	Effects on plan	Applicability tests
'expmethod'	This PMF is used in conjunction with decisions for the 'choosexp' choice type. If the expansion method chosen is not "use an operator", then it will change the type of the node being expanded from 'goal' to 'phantom'.	If the node to be expanded is of type 'action', then the only expansion method that will be considered is "use an operator". If the node is of type 'goal', then the three available expansion methods can be used, cf. "allow node to be already satisfied", "make links to satisfy", or "use an operator". A test is made on the "allow node to be already satisfied" decision, to make sure that the fact actually does hold at the node.
'operator'	The decisions available at a 'chooseoper' choice will come from the 'operator' PMF definitions, which are compiled, before the commencement of planning, from the user specified Task Formalism.. An operator can affect all three of MODPLAN's plan representation structures. It can add new nodes, which will affect the plan network, it can introduce conditions at the node being expanded, which will affect 'Gost', and it can give rise to effects, which will affect 'Tome'.	The system tests the 'holds' conditions of an operator, rejecting it if they are not satisfied.
'contributor'	This is for use with the decisions associated with 'choosecontrib' choices. If a contributor that needs linking is chosen, then the appropriate link is made in the network. Otherwise, there is no effect on the plan.	If the contributor has to be linked to the node at which a fact is to be made true, make a loop check on that link.
'instfact'	This is used in the decisions for a 'choosefact' choice. They have no effects on the plan.	No tests needed.

PMF name	Effects on plan	Applicability tests
'instantiate'	This PMF is used in the decisions for the 'chooseinst' choice type. When one of these decisions is chosen for execution, the system will insert a condition into 'Gost' if there is not already one there for the fact being instantiated. The condition thus inserted, or the already existing condition, will then have a new set of contributors for the new instantiations added onto it's existing set.	The constraints on the possible values of variables are checked before instantiated facts are accepted. These constraints are initially set by the user in the operator definitions, and help to prune the search in the solution space.
'interaction'	This is used with 'chooseint' choice types. It has no effect on the plan.	No tests required.
'correctint'	This is used with the 'choosecorr' choice type. There are seven interaction correction methods available, and for each one, a separate piece of 'correctint' PMF is defined. All of the interaction methods involve either making links, breaking ranges, or both.	Make loop checks for those methods that involve linking. Check that any methods which remove contributors to conditions do not remove *all* contributors if the conditions are of type 'supervised' or 'holds'.
'unsupervised'	'unsupervised' PMF is used with the decisions generated for the 'choosecond' choice-points. It has no effect on the plan.	No tests need be made.
'satisfy'	This is to be used by 'choosesatis' choices. There are two definitions for this PMF. One is used for decisions that allow 'unsupervised' conditions to be "already satisfied". The second is used for decisions that allow 'unsupervised' conditions to be satisfied by putting additional links into the network. Neither of these has any effect on the plan.	If the satisfaction method to be used is "allow condition to be already satisfied", then check to see if it actually is satisfied, and reject it if not. If linking is to be used, then no tests are done.

PMF name	Effects on plan	Applicability tests
'stop'	This PMF has no effects on the plan associated with it. It is used in the generation of a decision for the 'stop' choice type.	No tests required.
'breakrange'	This PMF is used by those 'correctint' PMF's that use the removal of the contributor to a condition (the breaking of a range that caused an interaction) as part of their interaction correction algorithm. If the "range" to be broken is actually a single node, then nothing at all is done. If the range terminates in a condition with only one contributor, then the range cannot be broken, unless the condition is of type 'phantom'. If it is of type 'phantom', then the contributor is removed, and the type of the node at which the condition applies changed from 'phantom' to 'goal'. Otherwise, the node corresponding to the head of the range is removed from the contributor list of the condition.	The tests used depend on the particular 'breakrange' algorithm. Checks may be made to see if the range contains only one node, to see if the condition involved is of type 'phantom', and to see if the condition has only a single contributor.

The reader is referred to appendix C for the PROLOG representation of these PMF definitions.

3.4.4. Generating Decision Lists

Generating decision lists is a two stage process. The system will first look at the information obtained from 'Cinfo' (the structure used to pass information between choices). It will extract any information relevant to building up a list of decisions, and produce a "kernel" decision list.

Then, the decision generator will look at the high-level PMF assertions in the PROLOG database. Each choice type has one or more pieces of high-level PMF associated with it (although in some cases, the PMF may not actually do anything). Each piece of PMF has a set of applicability tests, which are used to find out if the PMF is applicable in the current plan.

For each decision in the kernel decision list, new decisions are generated, one for each piece of high level PMF for the current choice type whose applicability conditions are satisfied. These new decisions form the decision list that is passed on to the choice making system.

3.4.4.1. Obtaining Kernel Decision Lists

The sources of the information used in generating kernel decisions lists will be discussed in this section. There are two such sources - from previous choices (via 'Cinfo'), and from the plan. The following table describes where this information comes from for each choice type, and how it is used to form a kernel decision list.

Choice tvne	Kernel decision list generating information source
'start'	No source information is required; a single "dummy" kernel decision is generated.
'choosexp'	A list of expandable nodes comes via 'Cinfo' from the last 'start' choice. For each node, a new entry is made in the kernel list of decisions.
'choosexm'	The node destined for expansion comes from the last 'choosexp' choice via 'Cinfo'. One kernel decision is produced.
'chooseoper'	The node to be expanded by the operator is passed to the decision generator from 'choosexm' via 'Cinfo'. One kernel decision is produced.

Choice type	Kernel decision list generating information source
'choosecontrib'	The node at which a fact is to be satisfied by linking is passed from either 'choosexp' or 'choosesatis' via 'Cinfo'. The QA system (see the description of NONLIN in chapter 2) is used to find all the nodes that would be contributors to this fact, if it wasn't for unwanted interactions. Then a kernel list of decisions is produced, containing one of these potential contributors per decision.
'choosefact'	The list of facts, from which a kernel list of decisions will be generated with one-to-one correspondence, can come, via 'Cinfo', from 'chooseinst', 'chooseoper', 'choosesatis', 'choosexm' or 'choosecontrib'.
'chooseinst'	The fact for which instantiations are to be made, plus the node at which the fact is to hold, are passed from 'choosefact' via 'Cinfo'. The question answering system is used to find all of the nodes in the network that could satisfy the fact. A kernel list of decisions is generated in which each decision represents possible instantiations that would satisfy the fact, and the nodes from which this particular set of instantiations came from.
'chooseint'	A list of interaction records may be generated by the last 'choosecorr' choice type, or by the last 'chooseoper' or 'choosecontrib' choice types. These records are passed via 'Cinfo'. For each record, a decision will be put into the kernel list of decisions.
'choosecorr'	The interaction to be corrected is used to form the single kernel decision; it comes from the previous 'chooseint' decision, via 'Cinfo'.
'choosecond'	A list of 'unsupervised' conditions to be satisfied will be passed to 'choosecorr' from 'start' via 'Cinfo'. The kernel decision list will contain one entry for each of these unsatisfied conditions.
'choosesatis'	The 'unsupervised' condition to be satisfied is passed via 'Cinfo' from 'choosecond', to form the single kernel decision for this choice.
'stop'	No information is used. A dummy 'stop' decision is generated as the kernel.

3.4.4.2. Obtaining the Complete Decision List

A complete decision list is generated by taking the PMF from each decision in the kernel decision list, and comparing it with the high-level PMF in PROLOG's database. Every time a piece of PMF with the same name is found, it's applicability conditions are tested. If they are satisfied, then a new decision is produced, using the found PMF for it's "action" part. This new decision is added onto a new list of decisions. When every piece of PMF for every kernel decision has been tested in this way, the final "new" list of decisions is treated as the complete list of decisions.

Frequently, only one piece of PMF will be found in the database. However, there are four cases in which more than one piece of high level PMF will exist for the current choice:

(1) If it is of type 'choosexm'. In this case, there are three possibilities- a fact may be already satisfied at a node, it may be possible to satisfy it by making links in the net, or it may be satisfied by choosing an operator to expand it. One piece of PMF will exist for each of these.

(2) If it is of type 'chooseoper'. Each operator definition, specified by the user in Task Formalism, will have been compiled by the system at the beginning of the planning process into a separate piece of PMF. These are used in generating decisions.

(3) If it is of type 'choosecorr'. There are seven possible interaction correction methods, each of which will have it's own PMF definition. These methods are more fully documented in the section on NONLIN, in chapter 2.

(4) If it is of type 'choosesatis'. Satisfying an 'unsupervised' condition can be done either by finding nodes in the net that already contribute to it, or by making links in the network to make it true. There will be a piece

of PMF for both of these possibilities.

The applicability tests used to decide which decisions can be incorporated into the decision list vary with the choice type currently under consideration. Their only purpose is to remove those decisions that should definitely *not* be considered, usually because they could lead to undesirable situations, such as loops in the plan network. The tests used by MODPLAN have been implemented to make the system retain the same basic choice making mechanisms as NONLIN. They are described in the table of high level PMF operators that appears earlier in this chapter.

3.4.5. User Interaction and Decision Selection

3.4.5.1. Introduction

Once a list of decisions has been generated, and passed by the failure detection mechanism, it possible to select one of these decisions for execution. Normally, the selection of a decision is an interactive process, ie. the user can control it. Interaction with the user is performed via an interactive system. MODPLAN also has an automatic mode, in which it can make any desired number of choices without user intervention.

3.4.5.2. User Interaction at Choice Points

3.4.5.2.1. Options Available from the Interactive System

The user is able to interact with MODPLAN at every choice-point, before the choice is made, as long as MODPLAN is not in it's automatic choice making mode. The user is presented by the interactive system with a menu, containing the following options:

(1) Instruct the system to select a decision. The system will use whatever algorithm is available to it for selecting a decision from the decision list. If the user has given any policy operationalisations or learned heuristics, then these will be used.

(2) Display the decisions available, and let the user make the choice. For each decision, the system will display the status (untried, succeeded or failed), the pattern from the PMF, and a "figure of merit". The figure of merit tells the user how highly the system, using automatic choice making algorithms, rated the decision.

(3) Force the system to abandon the current line of planning, and restart somewhere else in the choice-point tree. The current branch in the choice-point tree is preserved, so that it is possible to return to this point, and continue planning from where it had been broken off. The refocusing mechanisms are discussed in more detail in chapter 4.

(4) Enter an automatic mode, in which the system will plan continuously for a given number of decisions, making all choices without user intervention. On entering automatic mode, the user can ask the system to do two things. By specifying an integer, N, the user forces the system to plan automatically for N choices. By specifying one or more choice types, the user can force the system to allow user interaction only at those choice-points. If the single choice type, 'stop', is specified, the planner will plan automatically right up to the completion of the planning process.

(5) Print out the current plan. Currently, the system will print out the list of network nodes (with redundant links removed), followed by the 'Tome', followed by the 'Gost'. An attempt is made to format these

structures, to improve readability.

(6) Print out the current choice-point tree. The tree is printed as a list of nodes; some attempt is made to format them.

(7) Change the way in which the system selects decisions. This option allows the user to enter the information extraction function net and operationalisation editors. These are used for changing the system's choice making algorithms. A fuller account of these facilities is given in chapter 5.

3.4.5.2.2. Implementation of the Interactive System

MODPLAN has an 'interact' function, which allows the user and the system to conduct a simple dialogue. This function is called at every choice-point, before a decision is selected, and one of it's principal duties is to guide the user in selecting a decision. It is a recursive function. At each level of it's recursion, it calls two functions, 'ask' and 'act'. 'ask' is given the text of a question, prints it out and accepts the user's reply.[1] 'act' performs any action that the user's reply might suggest. A structure is passed down the recursion, to be modified by the 'act' functions. A question/answer session with the 'interact' function will thus have the net effect of transforming this structure into a new structure.

As an example, if the initial structure was a list of decisions, then a question/answer session for selecting a decision would result in a final structure that was a single decision. If the initial structure was the choice-point tree, then a question/answer session to find a choice-point on which to refocus attention would lead to a structure that was a single choice.

[1]As far as 'ask' is concerned, a menu is just a rather large chunk of text that has to be printed out as a question.

The 'interact' function has a couple of convenience features that make it a bit more complicated than the above description may suggest. These are:

(1) Various "special" questions that can be asked by the user. These are mainly calls to PROLOG functions not normally available to MODPLAN and commands that cause backtracking up 'interact's recursion. In the first case, once the "special" commands have been executed, we want the system to erase all evidence of them. The simplest way to do this is to use backtracking. To stop PROLOG from backtracking all of the way out of the interactive system, and failing the current choice, a system of ***failure traps*** has been implemented.

(2) The ability to accept more than one reply at once. If the user, on being asked a question, enters a sequence of command strings, separated by spaces, then the 'interact' function will suspend the printing of questions, and treat the sequence as a set of replies for the 'act' function to deal with. Thus, for instance, if the user typed 'auto' from the initial user menu, the interactive system would then ask for the number of choices to be made automatically. If the user had typed 'auto 5', then the system would simply go ahead and make five choices automatically, without asking any more questions.

It is the 'act' function that is at the heart of the interactive system, and this function is defined by a (large) set of assertions in PROLOG's database. An 'act' function is selected by a pattern match with the user's last reply. If a user types in an inappropriate reply, then 'act' will force the interactive system to backtrack to the previous question, and ask it again. Each reply belongs to a class of replies; for instance, a reply 'sys' ("let system choose a decision") would belong to the class 'choose' (for "choose decision"). The reply class is set automatically by 'interact'. If a user reply is not in the current class, it is rejected by 'act'. Each 'act' definition specifies the text of the next question to be asked, and a function to be applied to the structure being passed down the 'interact' recursion.

3.4.5.2.3. Invisible Options

A number of "invisible" options are available from within the interactive system. These are classless, and can be called at *any* question. They include the ability to turn on PROLOG debugging, editing and reconsulting files, and entering a PROLOG break level. The following is a complete list of these options.

(1) Quit planning altogether. This option forces a PROLOG abort, taking the user back to the PROLOG interpreter.

(2) Turn on PROLOG's debugging mode. Any MODPLAN functions that the user has put PROLOG spy points on will act as entry points for the PROLOG debugger. Typing 'n' from within the debugger turns debugging off again.

(3) Enter PROLOG break level. Useful if the user wishes to execute a number of PROLOG commands. ^Z returns control to the program.

(4) Edit a file. This option calls the UNIX text editor, 'vi'. It can be used in the normal way. Exiting the editor returns control to the program.

(5) Exit a single PROLOG command. This can be used for doing things like consulting files from within MODPLAN.

(6) Go back to last question. This forces the interactive system to backtrack to the last question that it asked.

(7) Go back N questions. Forces the interactive system to backtrack over the last N questions that is has asked, where N is an integer specified by the user.

(8) Return to first menu. This forces the interactive system to backtrack all the way to the user menu that was first printed out when the interactive system was entered.

(9) Log the creation times of choices. The time at which a choice-point is created can be used by the refocusing system. However, since what we want is *real* time, rather than execution time, it is necessary to use the UNIX 'date' command, pipe it's output to a file, and then read the file into PROLOG. This is messy, and time consuming, and MODPLAN's default is not to do it at all.

(10) Turn off the logging of choice-point creation time. Undoes the effect of the above.

(11) Suppress the printing of menus. This causes only the headings of menus to be printed out; the available options are not printed. Only really useful to an experienced user, when the system is running slowly.

(12) Remove the suppression on the printing of menus. Undoes the effects of the above.

3.4.5.3. Automatic Decision Selection

A decision will be selected automatically from the decision list at the current choice-point if either:

(1) The user had requested the system to make a choice from within the interactive system, or

(2) MODPLAN is running in automatic mode.

Selecting a decision is done as follows. First of all, all decisions with status 'fail' are removed from the decision list. Then, a function called 'merit' is applied to each decision in the decision list. This function will find the decision grading algorithm for the current choice type, execute it with a decision as input, and then instantiate the 'Merit' argument of the decision to the merit figure produced by the algorithm. Once all of the decisions have been assigned a figure of merit in this way, the system picks the one with the *highest* figure. The decision grading algorithms are discussed in more detail in chapter 5.

3.4.6. How MODPLAN Deals With Failures At Choice Points

3.4.6.1. Introduction

Whenever a decision list is created or changed, it is tested by a failure detection algorithm. If the list does not pass this test, then the current choice has *failed*. MODPLAN will backtrack to the previous choice, and try again. There are two complications, however. First, although MODPLAN uses PROLOG's backtracking mechanism, some method must be provided to stop backtracking when the appropriate choice-point has been found. Second, the information about which of the decisions at the backtracked-to choice-point lead to the failure will not have been preserved. If nothing were done about this, the decision might be selected again. Both of these difficulties are overcome by using a failure trapping system, discussed in a later section.

PROLOG runs under a backtracking régime. If the programmer makes a function non-deterministic (by not using "cuts"), PROLOG will try each clause for the function, as they appear in the database. If all of them fail, then PROLOG will backtrack to the most recently tried previous non-deterministic function, and continue from there. An exhaustive (but unintelligent) exploration of all possibilities at each choice-point, simply using PROLOG's backtracking mechanism, *could* be used by a planner. However, there are a number of reasons for wanting to augment PROLOG's backtracking capabilities in the case of MODPLAN, including:

(1) We wish to preserve the complete tree of successful choices. This is to be used by the refocusing mechanism (see chapter 4). The method used for refocusing is to force the system to backtrack to an appropriate point in the tree. Then we plan forwards, using the choices stored on the appropriate branch of the tree, until the choice-point to which focus is to be changed to is found. Since this tree is passed as an argument from one level of the 'plan' function's recursion to the next, straightforward backtracking would destroy that branch of the tree which we are abandoning. Thus, whenever the *user* wishes to force backtracking, the current choice-point tree must be asserted in PROLOG's database, to be retrieved at the target choice-point.

(2) At a given choice-point, we wish to preserve all of the available decisions, including those that led to failure. This information could be useful if this choice-point is ever backtracked to, to prevent such decisions from being selected again, and, perhaps, to improve the selection procedures used with the remaining untried decisions. If PROLOG's backtracking mechanism had been used, decisions that had failed would never be available for examination.

3.4.6.2. Failure Detection

The failure detection system is brought into action between the creation or changing of a decision list, and the making of a choice. It performs a set of tests over the *entire* list of decisions. If the decision list passes these tests, then MODPLAN moves on to the task of decision selection. Otherwise, the choice as a whole is failed, and MODPLAN backtracks to the previous choice. For each choice type, the failure detection tests will be specified in an assertion in PROLOG's database, called 'neededinfo'.

The detection of failures is one of the three (separate) components of a function called 'decimplement. This function is recursive. It tests a decision list for failure, selects a decision if this test succeeds, and then tries to execute the decision. If execution is successful, then MODPLAN moves on to the making of the next choice. If not, then the decision is given a status 'fail', and the updated decision list fed into a new level of 'decimplement's recursion. Recursion terminates when either a successful decision is selected, or the decision list becomes unviable.

The following is a list of MODPLAN's choice types, with their associated failure detection algorithms:

Choice Type	Failure detection algorithm
'start'	None
'choosexp'	All decisions have status 'fail', or the decision list is null
'choosexm'	All decisions have status 'fail', or the decision list is null
'chooseoper'	All decisions have status 'fail', or the decision list is null
'choosecontrib'	All decisions have status 'fail', or the decision list is null
'choosefact'	All decisions have status 'fail'
'chooseinst'	All decisions have status 'fail', or the decision list is null
'chooseint'	One or more of the interactions are for conflicts with facts specified in the Task Formalism as being always true, or all decisions have status 'fail'
'choosecorr'	All decisions have status 'fail', or the decision list is null
'choosecond'	All decisions have status 'fail'
'choosesatis'	All decisions have status 'fail', or the decision list is null
'stop'	None

3.4.6.3. Passing Failures Back

3.4.6.3.1. Failure Trapping

Failure can occur under two circumstances - either during the refocusing of the attention of the planner, or when the failure detection algorithm fails a choice. In both cases, PROLOG will backtrack, and, with a completely deterministic program, would backtrack right out of the program. To avoid this, MODPLAN sets failure traps. The traps used for the two types of failure are slightly different in detail, but the principle is the same. The trapping mechanism for choice failure will be described.

During planning, the system maintains a *trail*. This is a list of choice-point numbers, in the order in which the choices were made, from the first choice to the choice currently being considered. If a failure occurs, the system looks at this trail and picks off the number of the *last* choice to have been made. An assertion is made in PROLOG's database, containing the last choice number, and a piece of PMF (whose purpose will be described in the next section). A failure is now forced.

The system uses a function called 'decmake' at each choice. 'decmake' has a two part definition. During normal planning, 'decmake' generates a decision list, selects a decision to be executed and so on - all the things that have been described in the last few sections, in fact. 'decmake' is not deterministic, however. When backtracked to, the second part of it's definition is called.

This part of it's definition tries to make a match with the failure trapping assertion in the database, using the number of the current choice-point as the key to this match. When the choice-point to be backtracked to is found, this match will succeed. The system will now attempt to remake this choice, ensuring that the decision that lead to failure has it's status changed to 'fail', so that it is not considered again.

3.4.6.3.2. Failure Propagation

During decision generation at a choice-point, each decision is given a status. Normally, the status arguments of the decisions will not be instantiated, indicating that that they have not been tried. However, if this choice-point had been backtracked to, as a result of a subsequent choice failing, it is necessary to instantiate the status of the decision that led to failure to 'fail'. Since we will have backtracked from the failing choice, we cannot pass the information about which decision caused the failure via a variable. So the assertion used in failure trapping is used.

The method is as follows. The first time that a choice is made, the PMF associated with the decision selected is passed on to the next choice. If this choice fails, then this piece of PMF, along with the number of the previous choice, is put into the assertion used by the failure trapping mechanism. The system backtracks to the previous choice-point, and passes the piece of PMF onto the decision generating function. This checks the PMF of each decision generated against the PMF known to lead to a failing choice. When it finds a match, it instantiates the status of the offending decision to 'fail'. This ensures that that particular decision is never selected by the automatic choice making algorithm.

3.4.7. Information Passing in MODPLAN

3.4.7.1. Introduction

The choices made by MODPLAN are not independent; the way in which one choice is made can directly affect several others, and indirectly affect many others. An information passing protocol was developed, to enable choices to pass on relevant information to dependent choices. Passing information can be split into two parts - packaging information and putting it into the information passing structure (for the use of dependent choices), and picking up information from previous choices.

MODPLAN uses a structure called 'Cinfo' for passing information between choices. It is rather similar in concept to the "blackboard" used by Hayes-Roth et al in their Hearsay II program ([ERMAN, 1980]), although it is somewhat less flexible than the blackboard. It is a list of 'decinfo' structures, each one of which carries a piece of information, plus an identification. This identification takes the form of a record of the source and target choices, plus a name, describing the *format* of the information. For example, the format name 'expandables' tells the system to expect a list of expandable nodes.

Each choice type has associated with lists of "available information" (ie. information that could be supplied by it), and "needed information" (ie. information needed before the choice can be made). These lists are stored in two assertions in PROLOG's database, called 'availableinfo' and 'neededinfo' respectively. They contain, for each choice type, information format names, with associated lists of the choice types from which this information is expected to come. For instance, 'start' choices need information on the outstanding expandable nodes and the outstanding unsupervised conditions. The 'neededinfo' entry for 'start' looks like this:

```
'by(start, [from([choosexm, chooseint], expandables),
        from([choosexm, chooseoper, choosecond], unsupervised)])'
```

Before making a choice, the planner will look in 'neededinfo', to find the format names of the pieces of information that will be required during choice making. It will then scan through 'Cinfo', to see if it contains any of these pieces of information. Those which are found are deleted from 'Cinfo', and used.

It is quite possible that, at this "information gathering" stage, an attempt to extract "needed information" from 'Cinfo' fails to get everything needed for the current choice type. In this case, the system must fall back on (possibly time consuming) standard algorithms for extracting the missing information from the plan.

After a choice has been made, the system compares the "available information" for the current choice with the "needed information" of dependent choices. The information associated with each information format name in the intersection of these two sets is found for the current choice type. This information is then put into 'Cinfo'.

3.4.7.2. What Kind of Information is Needed at Each Choice Point?

When a new choice is to be made, a certain minimum amount of information must be available to guide the decision generation and selection processes. The system will use the 'curinf' ("current information") function to obtain this information. There are two sources of information used; the 'Cinfo' structure, that passes information from previous choices to the current one, and the plan. 'curinf' will look in the 'neededinfo' assertion, to find out which information is needed at the current choice. It will then search for this information in 'Cinfo'. If any of the needed information cannot be found here, then standard functions are invoked to extract it from the plan. The following table gives, for each choice type, the information needed to make the given choice, and an indication of where the information comes from:

Choice type	Information extracted from plan	Information extracted from 'Cinfo'
'start'	None	The lists of expandable nodes and 'unsupervised' conditions are needed in deciding the type of the next choice, and for passing onto other choices.
'choosexp'	None	The list of expandable nodes is needed to generate a decision list at this choice type.
'choosexm'	None	The number of the node to be expanded, plus the pattern and value on that node are needed in generating the decision list. The list of expandable nodes is needed, to be updated and delivered via 'Cinfo' to subsequent choices needing this information. The list of 'unsupervised' conditions is needed, and passed on unchanged to a subsequent choice.

Choice type	Information extracted from plan	Information extracted from 'Cinfo'
'chooseoper'	None	The number of the node to be expanded, plus the pattern and value on it are required, so that the system can use pattern matching to select suitable operators, and so that it knows where to insert the expansion in the network. Since an operator may introduce new expandable nodes into the network, the list of expandable nodes is needed, so that it can be updated and then passed via 'Cinfo' to subsequent choices. Unsupervised conditions may also be generated by an operator, so the list of 'unsupervised' conditions must be passed to the current choice, any new 'unsupervised' conditions added on, and the new list put into 'Cinfo'.
'choosecontrib'	The questioning answering system is used to find all potential contributors to a fact at a node.	The node number, and the pattern and value to be satisfied at it are required by this choice type.
'choosefact'	None	A list of the facts tc be instantiated is needed for generating a decision list. We also want to know what choice started the current sequence of 'choosefact'/'chooseinst' choices off, in order to decide on the type of the next decision.
'chooseinst'	The question answering system is used to find all critical nodes for the fact at the node at which instantiations are to be made. All nodes which are critical for identically instantiated facts are bundled into their own list and a list of these lists of contributors is generated. This is the basis of the decision list for 'chooseinst' type choices.	The fact to be instantiated, plus the number of the node at which it is to be satisfied are needed.

Choice type	Information extracted from plan	Information extracted from 'Cinfo'
'chooseint'	None	A list of interaction records is needed to generate the decision list and decide on the type of the next choice. This might be in the correct format, if the previous choice was of type 'choosecorr', or it might be expressed as a list of interacting ranges, if the previous choice was of type 'chooseoper' or 'choosecontrib', in which case a list of interaction records must be generated. The list of expandable nodes is also needed at 'chooseint' choice-points, to be passed on either to the next 'choosecorr' choice, if there is one, or to 'start'.
'choosecorr'	None	The interaction to be corrected is needed, to be used as the basis of a decision list. A list of expandable nodes is also required, to be updated if any 'phantom' nodes are converted into 'goal' nodes by interaction correction. The updated list is passed back into 'Cinfo' for the use of future choices.
'choosecond'	None	A list of 'unsupervised' conditions, passed from the last 'start' choice, is used to produce the decision list.
'choosesatis'	None	The pattern and value of the 'unsupervised' condition to be satisfied, plus the plan node they are to be satisfied at, are needed, to generate a list of decisions.

3.4.7.3. What Kind of Information is Available at Each Choice Point?

At choice-points of different types, different kinds of information will become available after a decision has been implemented. Some of this information is essential if future choices are to be made properly. The rest will be information that could be obtained by future choices from the plan. To avoid the duplication of effort, however, it would be better if this information, plus the essential information, of course, were put into 'Cinfo', where it would become available for use by subsequent choices. The function 'store' does the job of deciding what information to put into 'Cinfo'. Using the table of needed information format names ('neededinfo') in the PROLOG database, it finds out what information might be needed by different types of subsequent choice. If it has information available for any of these choices, it generates 'decinfo' structures for each of the choices, putting the available information into these structures. The list of new 'decinfo' structures is appended onto 'Cinfo', to produce an updated information passing list.

Here is a table of the kinds of information that become available at different choice types, and which choices that information is passed on to:

Choice Type	Available Information	Where it Goes
'start'	List of expandable nodes	'choosexp'
	List of 'unsupervised' conditions	'choosexp'
'choosexp'	Node to be expanded	'choosexm'
	List of expandable nodes	'choosexm'
'choosexm'	Node to be expanded	'choosefact' 'chooseoper' 'choosecontrib'
	If the expansion method selected was "allow node to be already satisfied", then a list of facts, containing the single fact to be satisfied, will be available.	'choosefact'
	If the expansion method selected was "make links in the network to make a fact satisfied", then the fact, with it's associated list of constraints on variables, becomes available.	'choosecontrib'
	List of expandable nodes	'start' 'chooseoper' 'choosecontrib'

Choice Type	Available Information	Where it Goes
	List of 'unsupervised' conditions	'start' 'chooseoper'
'chooseoper'	List of facts for instantiation	'choosefact'
	List of interacting ranges	'chooseint'
	List of expandable nodes	'chooseint'
	List of 'unsupervised' conditions	'start'
	Type of current choice (ie.'chooseoper')	'choosefact'
'choosecontrib'	List of facts for instantiation	'choosefact'
	List of interacting ranges	'chooseint'
	Type of current choice (ie.'choosecontrib')	'choosefact'
'choosefact'	Fact selected for instantiations, along with it's associated list of constraints on variables	'chooseinst'
	Updated list of facts for instantiations	'choosefact'
	Type of current choice (ie.'choosefact')	'choosefact'
'chooseinst'	None available	
'chooseint'	Interaction record	'choosecorr'
	Updated list of interaction records	'chooseint'
	List of expandable nodes	'choosecorr'
'choosecorr'	Updated list of expandable nodes	'chooseint'
'choosecond'	The fact involved in the 'unsupervised' condition, plus a list of the constraints on it's variables is available.	'choosesatis'
	Updated list of 'unsupervised' conditions	'start'
'choosesatis'	If the method chosen for satisfying an 'unsupervised' condition is "allow to be already satisfied", then a list of facts for instantiation, containing only the fact satisfied, is available.	'choosefact'
	If the method chosen for satisfying an 'unsupervised' condition is "make links in the network that will make the fact true", then the fact, along with it's associated list of variable constraints, is available.	'choosecontrib'
'stop'	None available	

3.4.8. Detecting Plan Completion

The planner recognises that the planning process is complete when there are no more expandable nodes left, and all 'unsupervised' conditions are satisfied. At this point, the planner does not stop, however. Planning cannot continue on the current branch in the search space, of course, but the 'stop' choice-point has all of the interactive options that are available at other choice-points. These include the ability to focus the attention of the planner to any other part of the tree and restart planning, so that alternative plans may be produced. The user also has the option of quitting MODPLAN completely.

3.4.9. Summary of MODPLAN's Operation

MODPLAN's operation is governed by a recursive function, called 'plan', which, at each level in it's recursion, makes a choice relating to the development of a plan. Before a choice can be made, at a given level of the recursion, a list of **decisions** must be generated, and then tested by the failure detection mechanism. If the decisions pass the test, then **choice making**, which consists of selecting one of these decisions, will occur. The system will execute the decision, and then work out the **type** of the next choice, and pass control back to the recursive 'plan' function.

The current version of MODPLAN "models" a non-linear planner called NONLIN. The types of choice made by MODPLAN are those found from an analysis of NONLIN. The order in which choices of different types are made forms part of MODPLAN's control structure, and is also based on NONLIN. The following algorithm shows, approximately, how this ordering of choice types guides the planning process in MODPLAN:

(1) MODPLAN is first given a snapshot of the plan (ie. the network, plus 'Tome' and 'Gost'). It is also given, or works out for itself, the list of expandable nodes for the current plan. If this is null, it will go to (4). Otherwise, it uses whatever policies and heuristics it has been provided with to select the most appropriate node to be expanded.

(2) A snapshot of the plan, plus the number of the node to be expanded, are passed on to a function which will choose an expansion method. This can decide that the pattern on the node is already satisfied, and that the node does not need any further attention. Or it may decide to make some links in the network, so that the pattern on the node becomes true. In either of these two cases, no more work has to be done, apart from removing the node from the list of expandable nodes. Control can be passed back to (1). The system has a third option, however. It can decide to use an operator to make the pattern true. Selecting an operator leads to making choices about instantiation of variables, and correction of interactions.

(3) Any interaction records generated by applying the chosen operator to the plan, plus a snapshot of the plan, will be passed to an interaction correction choice maker. An interaction is selected for correction, and one of the 7 possible methods for correcting interactions is chosen and implemented. This will involve making links in the plan network, and/or removing contributors to conditions. Another interaction can now be selected for correction, be corrected, and so on, until there are no outstanding interactions left. Now the system returns control to (1).

(4) If there are no more expandable nodes left, the system checks to see if any 'unsupervised' conditions have been introduced by any of the operators applied during planning. If not, then it goes straight to (5). Otherwise, the 'unsupervised' conditions must be satisfied.

(5) Planning is complete; the system stops. The user can interact with the system at this point, and is able to refocus attention to some other point in the choice-point tree, or quit planning entirely.

CHAPTER 4

Refocusing the Attention of a Planner

4.1. Overview

Since choices are stored in a tree, there is obvious potential for moving about in the tree, so that planning can restart at any point, and so that alternative plans can be generated. In this chapter, the interactive facilities available to the user for refocusing MODPLAN's attention are discussed.

There are four important motivations for wanting to be able to refocus the planner's attention:

(1) If a plan has been successfully generated, the user may be interested in looking at alternative plans, perhaps because the current plan contains redundant actions, or is not efficient enough in it's use of resources.

(2) The user may feel that the current direction of planning is unsatisfactory, maybe because the system is thrashing about, doing a lot of work that would not have had to be done if a better choice had been made earlier on, or on a separate branch of the choice point tree.

(3) If there are bugs in the domain description or the planning program, some branches of the choice point tree may lead to failure, and it could

be very irritating to have to start planning from scratch again.

(4) The user may wish to use a given set of choices as examples for a heuristic learning program; refocusing facilities could make it much easier to find the desired choices.

Rather than presenting the user with the entire choice point tree to peruse, the system allows the specification of various pieces of information about the target choice point, such as which plan nodes it affected, whether it was involved in interaction correction, etc. The system finds all choices which match this information. The more information the user supplies, the fewer choices will match. Selecting a target choice from this subset of the available choices would be easier than selecting from the complete tree.

4.2. System Components

4.2.1. Introduction

The restarting aids work using:

(1) An interactive system, that allows the user to engage in a simple dialogue with the program, in which the possible choice-points that could be jumped to are gradually narrowed down to one. This is the same interactive system as described in chapter 3. The plan restarting aids can be accessed from the menu printed out at any choice point.

(2) A set of functions, which employ user-specified features of the desired target choice-point to extract subsets of the full list of choice-points. By chaining several different functions of this type, one after another, the number of choice-points that the user could possibly jump to is rapidly reduced to a manageable number.

(2) is discussed in more detail in the next section. The interactive system (mentioned in (1)) needs a number of pieces of information to be able to carry out the restarting options. First, it needs the **choice-point tree'**, for those functions that involve tracing backwards or forwards along branches of the tree (eg. the function that finds the ancestors of a given choice). The choice-point tree is not changed by the interactive system. Second, a list of **available choices** is needed. Initially, this list will contain the whole of the choice-point tree. The plan restarting system is intended to work, however, by an incremental elimination of possible choices, until only one choice (the target) remains. The choice extracting functions will remove a subset of the given list of choices, and this subset will be passed onto the next 'interact' function as the new list of available choices. Those functions which trace backwards or forwards in the tree will form the intersection of the choices got from the tree with the list of available choices.

For instance, we may want to find all the ancestors of a choice N that are of type 'choosexp'. We could use the function that finds all choices of a given type to extract all of the 'choosexp' choices from the list of choice-points for us. Then we could apply the function for finding the ancestors of a choice. This traces forwards through the choice-point tree, to find *all* the ancestors of N. Then, it finds the subset of these ancestor choices which are of type 'choosexp', by forming the intersection of the previously found set of 'choosexp' choices and the set of ancestors.

There is a "negate" option available to the interactive system when restarting options are being used. This works in a fairly simple minded way. If a choice filtering function is preceded by 'n', then the system will first extract choices from the list of choices in the normal way for that function. Then, it will find the *set difference* between the choice list originally supplied and the extracted list. It will use this difference as it's output. In other words, all choices that the given function would normally *not* extract are found.

It should be noted that the "negate" option works in a different way from normal for some of the functions described below. For instance, there is a function which will look for all choices at which a given condition type is mentioned; the negation of this would make the system find all choices at which a condition type was mentioned, as long as it was *not* the given condition type. These special cases are discussed in a later section.

The functions which filter out choice-points can be grouped into five different classes, corresponding to the nature of the information they key on; any of the choice-point arguments, except the parent node, can be used as a key for the filtering of choices. These functions are described more fully in the following sections. The structure of the choice-point tree is discussed in chapter 3.

4.2.2. The Choice Filtering Functions

4.2.2.1. Filtering Functions Keying on Choice Number

All of these functions use the choice number arguments of the choices in the choice-point tree to select a target choice. The following options are available:

(1) Single step back. This finds the parent of the current choice, and refocuses attention on that. This, and the "single step forward" option, are probably the most commonly used of the refocusing aids. They can be used together, for doing things like changing the label on a previous choice, and then returning to the original choice.

(2) Single step forward. This finds the *first* child of the current choice, and refocuses attention to it.

(3) Jump N choices back. Refocuses on the choice N nodes back along the current branch in the choice-point-tree.

(4) Jump N choices forwards. Refocuses on the choice N nodes forwards on the current branch of the choice-point tree. If there are branches, this function will always select the *first* branch.

(5) Jump to absolute choice number. This refocuses attention on the choice whose number is supplied by the user.

(6) Find all choices local to the current choice. This means that the parent, siblings and children of the current choice are extracted from the list of choices supplied to the function, if they can be found. This is only really useful in branchy parts of the choice-point tree.

(7) The "next" choices are found. This extracts the children of the current choice from the list of choices supplied.

(8) The "descendants" of the current choice are found. This causes the system to extract all of the ancestors (ie. the children, the children of the children, etc.) of the current choice from the choice list. This is useful when the user is interested in examining all of the choices consequent on a given choice.

4.2.2.2. Choice Filtering Functions Keying on Choice Type

If the user knows the *type* of the choice to which attention is to be refocused, then one of the following functions would be used:

(1) Jump back to the last choice of a given type. Given a choice type, the system will search backwards through the choice-point tree until it

encounters a choice of that type; attention is refocused on that choice. Thus, for instance, if the current choice was of type 'chooseint', and the user felt that there were too many interactions to correct, then the system could be forced to refocus on the last 'chooseoper' choice, and a new operator selected. This, hopefully, would introduce fewer interactions.

(2) Jump forwards to the next choice of a given type. Given a choice type, the system will search forwards through the choice-point tree until it encounters a choice of that type; attention is refocused on that choice. If there are any branches in the tree, the system will select the *first*. This could be a useful feature if the user is re-examining a previously generated branch of the choice-point tree. For instance, the user could use this option to jump from one 'choosexp' choice to the next, all the way up the branch, to see how nodes had been selected for expansion.

(3) Jump back past N choices of a given type. Similar to (1), except that the searching process is repeated N times.

(4) Jump forwards past N choices of a given type. Similar to (2), except that the searching process is repeated N times.

(5) Find all choices of a given type. Given a choice type by the user, this function will extract from the list of choices all choices that are of that type.

4.2.2.3. Choice Filtering Functions Keying on Creation Time

As already explained in the section on the "invisible options" available from the interactive system, it is possible to make MODPLAN log the creation times of choice-points in the tree. One function is available to take advantage of this - find all choices made about T minutes ago, ± 30s. This function looks through the list of choices presented to it, and extracts all choices whose creation times are within a one minute band of the difference between the time at which the function was invoked and T. This function does not work well with choices that have been created during planning under the automatic mode, since a lot of choices can be made in a very short time. If the default "don't log choice creation times" option has been left on, then this function will do nothing.

4.2.2.4. Choice Filtering Functions Keying on Labels

One of the options that the user is given during the making of the choice is to "name" that choice, with a label. This means that the user can identify choices that may be of special interest at some future date. For example, if the user was not sure how good the selection of a decision is at a given choice-point, then this point can be labelled. It may transpire that, at some later stage of planning on the same branch of the choice-point tree, a bad choice had been made earlier on. An obvious choice for refocusing to, to restart planning, would be the labelled choice. Labels must be atomic, ie. strings of alphabetic and numeric characters. These strings are arbitrary, and there is nothing to stop the user from using the same label on more than one choice. It is recommended that some sort of convention is used in labelling nodes, eg. suspected bad choices could have labels starting with the string 'bad', etc. By default, choices will be labelled with the string 'nolabel', which is ignored by the functions that are described below. The following functions, then, can be used to refocus MODPLAN's attention:

(1) Jump back to nearest labelled choice. This will search back through the choice-point tree, until a labelled node is encountered, and refocus attention onto it.

(2) Jump forwards to nearest labelled choice. This will search forwards through the choice-point tree, until a labelled node is encountered, and refocus attention onto it. If the tree branches at any point, the ***first*** branch is taken.

(3) Jump to a choice with a given label. Given a label, the system will scan the entire list of choices until it finds a choice with that label, and then refocus attention on it.

(4) Find all labelled choices. This function extracts all labelled choices from the list of choices given to it.

(5) Find all choices whose labels contain a given string. The user enters a string of characters, and the system searches for, and extracts, all choices with labels containing this string. This function comes into it's own if the user has adopted a labelling convention of some kind. For instance, if the user had labelled all choices that were expected to lead to failure with 'fail1', 'fail2', 'fail3', etc., then all of these choices could be extracted by using this function with the string 'fail'.

4.2.2.5. Choice Filtering Functions Keying on Information in Decision Lists

These are the most exotic choice filtering functions. They look at the decisions in the decision lists stored at choice-points. They come in three flavours (indicated in **bold**):

(a) Functions which key on plan state information.

PMF, in decisions in the decision lists at choice-points, often contains plan state-related information, such as network node numbers, patterns, network node types and condition types. The following functions enable the user to take advantage of this fact in selecting a target choice.

(1) Find all choices for a plan node N. This function looks at the PMF in all the decisions for all of the choices in the current choice list. Any choices with at least one decision containing a reference to the node N are extracted. This is best used in conjunction with other functions, since the number of choices referring to a given plan node could be very large!

(2) Find all choices for a plan node type. This function looks at the PMF in all the decisions for all of the choices in the current choice list. Any choices with at least one decision containing a reference to the specified plan node *type* are extracted. Useful if, for instance, we wanted to find out whether a given plan node had at any stage become a 'phantom'.

(3) Find all choices for a condition type. This function looks at the PMF in all the decisions for all of the choices in the current choice list. Any choices with at least one decision containing a reference to the specified condition type are extracted.

(4) Find all choices for a pattern. This function looks at the PMF in all the decisions for all of the choices in the current choice list. Any choices with at least one decision containing a reference to the specified pattern are extracted. These patterns may be patterns on plan nodes (eg., to use a blocks world example, 'putontopof(a, b)'), or patterns in conditions and effects (eg. 'cleartop(c)'). This function will nearly always be used in conjunction with other functions, since a given pattern may be mentioned many times.

(5) Find all choices for a pattern/value combination. This is similar to the above,. except only those patterns having a user specified *value* are extracted (eg. 'on(b, c)=false'). For instance, we may wish to find all choices at which conditions for a given pattern/value combination are inserted into the 'Gost'. We can use this function to find *all* choices

involving this particular pattern/value combination. Then, we use the function which extracts all choices that introduce conditions into 'Gost' (described a little further on) to obtain the desired set of choices.

(b) Functions which key on plan modification information.

These functions use the *names* of the PMF found in decision lists to identify desired choices. This allows the user to find choices at which specific changes were made to the plan, since the purpose of PMF is plan modification.

(1) Find all choices at which plan nodes were created. This function looks at the PMF in the decision lists of all the supplied choices, and extracts those choices containing PMF with name 'makenode', the node inserting function. Thus,we could use this function, along with the function that finds all choices mentioning a given plan node, to find the choices at which a given plan node is inserted into the network (there may be more than one such choice, since the choice-point tree may have several branches).

(2) Find choices at which conditions were inserted into 'Gost'. This function looks at the PMF in the decision lists of all the supplied choices. Any choices containing PMF with name 'makecond' are extracted.

(3) Find choices at which effects were inserted into 'Tome'. This function looks at the PMF in the decision lists of all the supplied choices. Any choices containing PMF with name 'makeffect' are extracted.

(4) Find all choices at which 'goal' plan nodes are changed to 'phantom'. This searches through the current list of choices for all those with at least one decision containing PMF of name 'changetype', with it's third argument specifying a change from 'goal' to 'phantom'. This enables the user to locate the choices at which expansions were made by the "allow

a fact on a node to be already satisfied" and the "allow a fact to be satisfied as a result of linking" methods.

(5) Find all choices at which 'phantom' plan nodes are changed back to 'goal'. This works in a similar way to the above, except in reverse. It is used for finding choices for interaction correction methods which remove all contributors to 'phantom' conditions.

(c) Functions which key on decision status information.

The 'Status' arguments of decisions can be used to identify choices containing them. This argument may be uninstantiated (if the decision has not been tried), instantiated to 'fail' (if the decision has been tried, and has failed) or it may have a value 'child(N)' (indicating that the decision had succeeded, and had resulted in the creation of a choice-point numbered 'N'). The following three functions use this 'Status' information.

(1) Find all choices with untried decisions. This looks through the list of choices supplied to it, and checks the decisions in the decision lists. Any choices with at least one untried decision are extracted. This is often a good first step if a plan has been generated and the user wants to try generating alternative plans.

(2) Find all choices at which branching has occurred in the choice-point tree. This looks through the list of choices for all those containing at least two tried non-failing decisions. This is a useful aid for users who wish to return to points at which there was some uncertainty about which direction planning should take.

(3) Find all choices at the tips of the choice-point tree. This looks for all choices in whose decision lists are only decisions with an uninstantiated 'Status' variable. This enables the user to find the points in the choice-point tree at which planning had previously been abandoned.

4.2.3. Negation of Choice Extracting Functions: Special Cases

As mentioned above, the negation of a choice extracting function does not always mean that the system will find all choices that would *not* be extracted by that function. The following lists the exceptions:

(1) Find all choices whose labels contain a given string. In this case, the negation would cause all choices whose labels did *not* contain the given string to be extracted; unlabeled choices would be ignored. Thus, for instance, if the choice-point tree had several branches, and all the labels on one of the branches started with the string 'branch3', then we could use the negation of this function to find all labelled choices not on 'branch3'.

(2) Find all choices mentioning a given plan node number. Negating this will cause the system to look for all choices at which a plan node number other than the given one is mentioned. This could be useful when it is known that the system has dealt with a given plan node correctly, and we want to investigate it's handling of *other* plan nodes in more detail.

(3) Find all choices mentioning a given plan node type. The negation of this looks through the 'Pattern' arguments of the PMF in the decisions of each of the choices presented to it, to see if they contain any strings falling into the set '[goal, action, phantom]', but not corresponding to the given plan node type. We could use this in situations where we are looking for choices that affect both 'goal' and 'phantom' plan nodes. To do this, we would use the negated plan node type hunting function with a type 'action'; this would find all choices that affect nodes, but do *not* affect 'action' nodes.

(4) Find all choices mentioning a given condition type. If this is negated, the system will use the same kind of algorithm as given for finding

choices associated with plan node types, except it will try to ensure that all strings found are in the set '[holds, supervised, unsupervised, phantom]'. As an example of it's use, we may want to examine all choices affecting conditions introduced by operators. To do this, we would simply use this negated option with a string 'phantom'. This works because all conditions, other than 'phantom' ones, are introduced by operators.

(5) Find all choices mentioning a pattern of given value. In it's negated form, this function will find all choices mentioning the given pattern with a value different from the specified value, ie. it looks for the converse of the given pattern/value combination. So, for instance, we may wish to find all choices at which effects are introduced which might interact with a given pattern/value combination. First, we use this negated function to find all choices containing patterns that are the same as the given pattern, but whose values are different from the value supplied. Then, we can use the function that finds all choices at which effects are introduced, to find the desired set of choices.

CHAPTER 5

Automating Choice Making

5.1. Introduction

Having discovered the choice types used by a planner, the next step is to provide the user and the system with facilities to help them to make *good* choices. For both system and user, some means of extracting information relevant to the current choice type is required. For the system, a formalism for representing the choice making algorithms is also desirable. An interactive facility is provided, which allows the user to filter and re-format information obtained from the plan, knowledge base, previous choices, etc. A standard set of functions is available to the user for building up procedures for filtering/reformatting information at each choice point. These procedures can be stored, so that requests by the user or the system for information will cause only the filtered/re-formatted information to be presented. The user determines empirically which information is most relevant at each choice type. A simple language, the "Operationalisation Formalism", can be used to build up algorithms for guiding choice making.

5.2. Policies and Heuristics in Choice Making

5.2.1. Introduction

In this chapter, a short excursion into automatic choice making will be made. Two classes of automatic choice making algorithm will be distinguished: those resulting from explicitly declared planning policies, and those resulting from learned heuristics.

5.2.2. Policies and Operationalisations

Policies are general strategies that the planner should follow, such as "minimise plan network linearity", that are specified explicitly by the user. Most planning systems do use various policies in the planning process, but these tend to be "hard coded" into the control structure of the planner. This makes it difficult for a novice user to follow the reasoning behind the choice making processes, and difficult for a more advanced user to change the planning policies. Wilensky ([WILENSKY, 1981]) advocates the explicit representation of meta-level themes, of which policies are a subset, in planning systems, and an attempt has been made to do this for MODPLAN.

Since the advice contained in policies is general, they are unable, by themselves, to guide the planning process. Mostow, in his work on FOO, coined the term "operationalisation" for the process of converting general advice into operational algorithms. In FOO, there are four classes of operationalisation technique, of which one, "goal specific operationalisation", is used by MODPLAN. The use of operationalisations in choice making was not one of the primary aims of the research on MODPLAN, so only the simplest operationalisation technique was considered.

This technique basically allows the user to specify how a policy should be operationalised. For a given policy, operationalisations can be specified for any number of choice types. In use, an operationalisation would, for a particular choice, examine each of the possible decisions and assign figures of merit to them. Since the operationalisations of several policies may affect a single choice type, MODPLAN provides a simple method of mediating between the conflicting preferences of these policies.

The following is a list of the policies that are being used with MODPLAN:

(1) Minimise plan network linearity.

(2) Find shortest path through solution space.

(3) Find a path through solution space not leading to failure.

(4) Maximise redundant contributors to a condition.

5.2.3. The Relationship Between Policies and Learned Heuristics

Learned heuristics are remarkably similar to operationalised policies. The system learns rules that make an assessment of the plan state and other information for each decision, and assigns to the decision figures of merit. It is proposed that these heuristics represent operationalisations of policies that are held in a person's head, but that are not explicitly stated. Of course, a person may use many policies, and the choice of a particular decision is a result of some mediation between several of these. It is likely, therefore, that such heuristics would not be easily understood, even by the person teaching them to the system. Thus, although a facility for learning heuristics would be useful, it would be sensible to try to get as many policies explicitly stated as possible.

In MODPLAN, the rules that form policy operationalisations are written by the user in a simple language, the Operationalisation Formalism (OF). This language has been found adequate for all of the policies so far considered, and it is proposed that it could also be used to represent the choice making rules generated by a learning system.

5.3. The Use of Policies in MODPLAN

5.3.1. Introduction

The way in which MODPLAN uses policies will be considered under three headings:

(1) Representation and operation. A formal representation for policies and the way in which they are operationalised is presented. Closely linked to this is the problem of how a policy operates. Here, a method for filtering relevant information and the "execution" of operationalised policies is considered.

(2) Modification. A pair of editors is described, which can create and modify policy operationalisations and their information filtering procedures.

(3) Storage. The means by which sets of policies are stored is explained.

5.3.2. Representation and Operation of Policies

In MODPLAN, a policy takes the form of a PROLOG structure, containing three arguments:

'policy(Name, Text, Operationalisations)'

The 'Name' of the policy is a string used by the program to identify it. The 'Text' is a quoted string. It enables the designer of the policy to put in some kind of humanly understandable explanation of what the policy is supposed to do. 'Operationalisations' is a list of PROLOG structures, each of which defines how the operationalisation is to be applied for a given choice type. If this list is null, then this particular policy will have no effect on planning. If it is not null, the structures in this list will each have five arguments:

(1) **Choice Type.** This is the type of the choice that the operationalisation defined in this structure will apply to.

(2) **Function Net.** This is a sub-net of functions that will extract the information needed by the operationalisation defined in this structure. It is a "sub-net" because it will be merged with the "sub-nets" for the other operationalisations for this choice type (if there are any), before the current operationalisation is actually applied.

(3) **Operationalisation.** This is the definition of the operationalisation itself. It is written in a simple language, called the "Operationalisation Formalism".

(4) **Weight.** This is a weighting factor by which the assessment that the operationalisation makes of a decision is multiplied. It is a number between 0 and 1, and is initially set to 0.5.

Policy application occurs when a choice is to be made, and works as follows. The system will be aware of the type of the current choice. Thus, it will be able to use pattern matching to select the correct operationalisation defining structure from the list contained in the policy definition. The system will apply the information filtering functions, found in the merged net for the current choice type, to the plan, decision list, etc. The filtered information so produced is then fed into the operationalisation, obtained from one of the other arguments of the operationalisation defining structure. In the light of this filtered information, the operationalisation is able to assign a figure of merit to each of the decisions in the decision list that was generated at the current choice.

The above process is repeated for all the policies that the user has defined, so that for each decision, there may be a number of merit figures.

A very simple arithmetic method is used to mediate between the conflicting preferences of these different policies. Each operationalisation has associated with it a weight - a number between 0 and 1. The figure of merit assigned to a decision by a given policy is multiplied by the weight. This weighted figure is added onto the weighted figures of merit from the other policies used in judging the decision. The sum of all these weighted merit figures is divided by the number of policies involved, to normalise it. We could express this in the following way:

$$m = (m_1w_1+m_2w_2+m_3w_3+\ldots\ldots m_nw_n)/n$$

where m is the overall figure of merit, $m_1,m_2,m_3,\ldots.m_n$ are the figures of merit obtained from each of the policies, $w_1,w_2,w_3,\ldots w_n$ are the corresponding weightings, and n is the total number of policies involved. The decision eventually selected by the system will be the one with the largest normalised figure of merit.

The next two sub-sections describe information function nets and operationalisations in more detail.

5.3.2.1. Information Function Nets

5.3.2.1.1. Introduction

In order for an operationalisation to work efficiently, it should not have to process all of the information available to it, ie. the plan, the list of decisions, the list of plan modification algorithms, and the list of information from other choices. MODPLAN has available a set of standard information filtering functions, and a structure for combining them (a function net). An editor enables the user to construct these nets of information filtering functions for each choice type. An interpreter, when supplied with the source information and the net, will evaluate the functions the net contains, producing a list of condensed information for the operationalisations used at the current choice.

5.3.2.1.2. The Function Net

This is a tool, designed to enable a user to combine information filtering functions, so that operationalisations can be supplied with the information that they require. There will be one net (stored in PROLOG's database) for each choice type. Notionally, it is a directed graph, with the nodes being information filtering functions, and the arcs carrying information from one node to another. These nets are created by the user using a function net editor, and they are "executed" by a function net interpreter. If the net is imagined to be directed from left to right, then on the left will be the nodes containing information source functions, and on the right, the information destinations, ie. operationalisations. Between these will be the graph of nodes containing information extraction functions. The overall effect of "execution" is to take the (large) amount of information available from the sources, and filter out everything except that information needed by the operationalisations for a given choice type.

The PROLOG representation for this net is a list of structures, corresponding to the nodes, each structure containing the following four arguments:

(1) **Node number.** A number, used to uniquely identify the node.

(2) **Function name.** The name of the information filtering function to be executed at the node.

(3) **Predecessors.** A list of the predecessor node numbers. The functions executed by these predecessors will supply the information needed at the current node.

(4) **Successors.** A list of the successor node numbers. Information from the execution of the current node's function will be passed to all of these successors.

5.3.2.1.3. Functions Available

In this section, we will look at how the information filtering functions are defined, and what functions are available at present. The definitions are complicated by the fact that they must contain not only an algorithm for filtering information, but also a specification of the input information needed by the function, and a specification of how to format it's output for printing.[1]

Information filtering function definitions take the form of assertions in PROLOG's database. There are three classes of information filtering functions:

[1] Printing the function's output is sometimes done during function net editing.

(1) **Information sources**. These are not actually executable functions, but to the system, they look as though they are. Currently, four sources of information are used: the plan representation (ie. plan network, 'Tome' and 'Gost'), the high-level PMF definitions, the decision list for the current choice, and 'Cinfo'.

(2) **Information extraction functions**. These take information from other information filtering functions, including information sources, and generate a condensed version of that information.

(3) **Information destinations** These correspond to the operationalisations needing the information. They are not executable functions - they just store information. They have no argument order specifications, since the information needed by an operationalisation is likely to change every time a user changes that operationalisation.

The information filtering functions contain the following five arguments:

(1) **Function Name.** This is the name of the filtering function. It should be an alphanumeric string, starting with a lower-case letter.

(2) **Argument Order.** This is a list of lists of function names. Each element in the top-level list corresponds, in it's position, to one of the arguments of the filtering function. These elements are themselves lists, which specify the *names* of the information filtering functions which may be used to supply the information for the corresponding argument.

(3) **Function Definition.** This is a PROLOG structure, whose arguments accept input information, specify a PROLOG function to be executed and pass back output information. The input information will be fed into the function, processed, and then leave via the output information argument.

(4) **Printing Format.** This contains a PROLOG function that can print the output of the function to the screen, in a user-readable format.

(5) **Description.** A piece of text (in English) describing what the function does.

A table of these information filtering functions is given below.

Function Name	Description
'source(plan)'	An information source function that gets the current plan.
'source(PMF)'	An information source function that gets the current high-level PMF from PROLOG's database.
'source(declist)'	An information source function that finds the current decision list.
'source(decinfo)'	An information source function that gets information from 'Cinfo' for the current choice.
'lastdecision'	Grade decisions in a decision list according to position in list, giving the last decision the highest rating, and the first decision the lowest rating.
'firstdecision'	Grade decisions in a decision list according to position in list, giving the first decision the highest rating, and the last decision the lowest rating.
'satorop'	Used with decision lists for 'choosexm' choice types only. Grade decisions by giving equal preferences to "already satisfied" and "operator", and a lower grading to "link".
'slop'	Used with decision lists for 'choosexm' choice types only. Grade decisions by giving maximum preference to "already satisfied", next to "linking", and minimum to "operator".

Function Name	Description
'mininst'	Used with decision lists for 'choosefact' choice types. This function grades decisions by counting the number of instantiations that would have to be made to their PMF 'Pattern' arguments for them to be fully instantiated. The maximum grading is given to the decision with the smallest number of uninstantiated variables.
'operationalisation(<policy name>)'	Information destination, associated with policy operationalisations.

5.3.2.1.4. Function Net Interpreter

The purpose of the function net interpreter is to take a large quantity of input information, and reduce it to a compact form that can be used by policy operationalisations and heuristics. It takes as input arguments the source information (currently the plan, the decision list and 'Decinfo', plus the PMF, extracted from PROLOG's database) and a network of information filtering functions. It produces as output a list of pairs, one for each operationalisation, containing information appropriate for each particular operationalisation.

It is supplied with an evaluated line of nodes and an unevaluated line of nodes. A "line" can be regarded as a list of nodes that would be in parallel in the net. Initially, the evaluated line is null, and the unevaluated line consists of the four information sources. The system attempts to evaluate each node in the unevaluated line. If it succeeds, then a pair, comprising the node number and the evaluation, is added to a new list of evaluated nodes, for the next level of the interpreter's recursion. An "evaluation" is the information found as a result of executing the function at a node. The criteria for evaluation success are:

(1) That all of the predecessor nodes of the unevaluated node are on the list of evaluated nodes.

(2) That, if (1) succeeds, then the function must execute successfully too.

If these criteria are not satisfied, then all of the nodes on the evaluated list that are predecessors of the failed evaluation are added to the new evaluated list. Having attempted to evaluate all of the unevaluated line, and generated a new evaluated line in the process, the system will then work out the new unevaluated line. To do this, it simply takes the successor lists for all the nodes in the new evaluated line, and forms the union of them. This information is dropped into the next level of the interpreter's recursion, and the process just described starts all over again. Execution is terminated when no new unevaluated line can be generated, because all of the nodes in the new evaluated line have null successor lists. Nodes without successors are called terminal nodes. During the interpreter's execution, any terminal nodes encountered are stored in a list, along with the evaluations of their predecessors. This is the information that will eventually be passed to the policy operationalisations, or to the user, if the interpreter is being used from within the function net editor.

5.3.2.2. Operationalisations in MODPLAN

5.3.2.2.1. Introduction

An operationalisation is the algorithm that expresses the specific application of a given policy at a given choice type. The information needed by the operationalisation comes from the evaluation of the merged function net for that choice type. This information takes the form of a list; the elements in this list contain information from selected functions in the function net.

The operationalisation definition consists of a list of **information processing functions**, one for each of the pieces of information passed from the function net, plus an arithmetic **expression**. This representation is known as the "Operationalisation Formalism", or OF. When an operationalisation is being run, each information processing function is executed in turn. The functions compare the decision currently being tested with the information in the list supplied, producing numerical assessments based on these comparisons. These assessments are passed to the expression, which produces an overall assessment for the current decision.

The user can change operationalisations, using the operationalisation editor. This editor can be called from the function net editor, and vice versa. The ability to switch in this way is vital, since changes to an operationalisation can affect the information that it needs, and changes to the function net can affect the information supplied to an operationalisation.

5.3.2.2.2. Operationalisation Formalism

Making operationalisations comprehensible to a user, and making them easy to change, is the purpose of the Operationalisation Formalism (OF). This is a declarative representation for operationalisations. In MODPLAN, a piece of Operationalisation Formalism would be used to judge each of the decisions in the decision list for the current choice in turn. A piece of OF comes in two parts:

(1) A list of information processing functions. Each function has associated with it the name of the function in the net from which it's information is to come, the name of a function that will compare that information with

the decision under scrutiny, and an assessment of that decision.

(2) An arithmetic expression. This expression relates the individual assessments from each of the information processing functions, to produce an overall assessment for a decision. A simple expression may produce the average of all the individual assessments; a more sophisticated method would be to find a weighted average. The operationalisation editor has an arithmetic parser that allows the user to enter expressions of arbitrary complexity. The results of evaluating this expression should fall between 0 and 5.[2]

Notionally, we can picture an operationalisation thus:

Net function 1	Processing function 1	A_1
Net function 2	Processing function 2	A_2
*	* *	
*	* *	
*	* *	
Net function N	Processing function N	A_n

Assessment = $f(A_1, A_2, \ldots\ldots A_n)$

$A_1, A_2, \ldots A_n$ are the assessments resulting from applying the processing functions 1...n to the information from net functions 1...n respectively, for a given decision. The function f describes the expression to be used to find the overall assessment for the decision.

The information (from the function net) used by an information processing function takes the form of a list of pairs, each pair being contained within a 'dinfo' structure, thus:

'dinfo(Decision, Info)'

[2]The original reason for adopting this particular range of values has since been found invalid, and it can be regarded as arbitrary.

'Decision' is one of the decisions in the decision list for the current choice. 'Info' is a numerical "grading" that has been assigned by the function net interpreter to the decision. 'Info' can take a real numbered value from 0 to 100.[3] For instance, the grading from a 'lastdecision' net function would relate each decision to a numerical indication of it's position in a decision list. This would be 100 if the decision is at the end of the list, 0 if it is at the beginning, and somewhere in-between for all other decisions.

Currently, there are only two functions available for comparing a decision with information from the net:

(1) Get the grading for the decision. This simply looks through the 'dinfo' structures from the net, until it finds a piece containing a matching decision. The grading associated with this decision is extracted, and forms the assessment for this function.

(2) Compare the grading for the decision with those of other decisions. This function can be given a comparison operator by the user, such as greater than, equal to, etc. It will find the grading associated with the decision in the 'dinfo' structures extracted from the net. It will then compare this grading with the corresponding gradings associated with each of the other decisions in the list of 'dinfo' structures. The nuber of comparisons which succeed form the assessment for this function.

5.3.2.2.3. Execution of Operationalisations

The purpose of the operationalisations that may be applied at a given choice type is simply to produce figures of merit for the decisions in the decision list at that choice-point. The system examines each decision in

[3]This range was selected to produce figures that look like percentages; it's function is purely aesthetic.

turn, applying all of the applicable operationalisations to these decisions. For each decision, a figure of merit will be calculated, by taking the assessments produced by each operationalisation, multiplying them by the appropriate weights, and then finding a mean value.

In order to execute the operationalisations for a given choice type, the system will first find the merged function net for that choice type. It will interpret that net, feeding in the required source information, and producing an output list, containing one element for each operationalisation. Each element in this list contains the information needed by a particular operationalisation.

In executing an operationalisation, the system executes each of it's information processing functions in turn. Since an information processing function knows the name of the the net function that should supply it with information, it is able to get this information from the output of the net interpreter. The processing function makes comparisons between the decision under test and this information, as described in the last section, and produces an assessment of the decision. Having executed each of the information processing functions for an operationalisation, the system finds the overall assessment, by using it's expression to combine the individual assessments from each of the information processing functions.

5.3.2.2.4. Operationalisations Used in Experiments

The operationalisations used in the experimental results chapter (chapter 6) are all very similar, and fairly simple. They take, as input, information from one of the information filtering functions in the appropriate merged function net. The 'getdinfo' information processing function is used to find the numerical assessment for the particular decision under scrutiny.

In all cases, the expression used just divides the 'getdinfo' output by 20. The reason for this is that all of the information filtering functions produce a grading on decisions, in the range 0 - 100. In order to make the final grading produced by the operationalisation fall between 0 - 5, this result has to be divided by 20.

Because it is possible to store all of the information relating to policies in files, the user is able to build up a set of policy operationalisation suites, in which the operationalisations for any given choice type may be varied. This means that the user can test out the effects of different operationalisations, or even different policies, on the final plan. This facility was used in producing the experimental results. An example of how a policy operationalisation suite looks, notionally, is presented in chapter 6. In appendix D, the actual PROLOG representation for the same suite is given.

5.3.3. Changing Policies

A representation for policies is of little value if the user of a system is unable to change them. In MODPLAN, it is not possible to create or destroy policies, without changing the program code. However, the user *can* change the way in which existing policies are operationalised.

This is done via MODPLAN's interactive system. The system gives the user a list of policies from which to select. For the policy selected, the system will allow the user to enter one of two editors, one for changing the operationalisations themselves, and the other for changing the information filtering function nets associated with them.

For a given operationalisation definition, the editors for its function net and operationalisation will be linked; the user will be able to move from function net editor to operationalisation editor, and back again. To quit this pair of editors, the user must use the 'save' option from either editor. This will save both the new sub-net and the new OF, generate a new "merged" net for the current choice type, and then return the user to the menu of policies for the current choice type. From this menu, it is possible to edit function nets and operationalisations for *other* operationalisation definitions.

5.3.3.1. Function Net Editor

The function net editor is a facility that was designed to enable a user to create new nets of information filtering functions, or to alter existing nets. It has options that allow the user to add, delete or change nodes in the net, display the current net, and interpret the net. The ubiquitous interactive system presents the user with a menu of these options, and passes the function net down it's recursion as the structure to be altered. The function nets that *can* be edited are confined to those in the operationalisation definitions for the current choice type.

The user is forced to think about the information needed by each operationalisation separately, by only being allowed to edit sub-nets for one operationalisation definition at a time. Once a sub-net has been edited, the system must do two things. First, the new sub-net, produced by the edit, must replace the old one in the operationalisation definition. Second, the sub-nets for each of the operationalisations pertaining to the current choice type, including the one just edited, must be "merged", and stored in PROLOG's database.

The following options are available from the function net editor:

Option	Description
Add a new node	Asks the user to supply a function name and a list of predecessor nodes. A new node is created, and numbered automatically. It is given a null successor list; this is automatically updated if the current node is ever mentioned as a predecessor by subsequently created nodes.
Remove a node	The node to be removed is mentioned by number. Predecessor and successor lists of other nodes are not altered.
Change a node	This option allows a user to change an existing node. There are three things that can be changed: the function, the predecessor list and the successor list.
Display functions	This displays the list of currently available information filtering functions. The system looks in the function definitions stored in PROLOG's database, and, for each one, displays the function name, and the text string found in the 'Description' argument.
Print Function net	The current function net is printed out as a list of nodes, displaying the number, function name, predecessor list and successor list associated with each node.
Operationalisation input	Print out the list of the information filtering functions required as input for the current operationalisation. These functions should form the final "line" in the function sub-net for this operationalisation.
Interpret net	Use the function net interpreter to find the information available to the current operationalisation at this choice point. The results are formatted and printed out on the screen.
Operationalisation Editor	This option takes the user into the operationalisation editor, described in a later section.
Save nets	This saves the edit for the current operationalisation and it's associated sub-net, and re-merges this sub-net with the sub-nets for the other operationalisations for the current choice type. The user is returned to the policy menu.

5.3.3.2. Operationalisation Editor

The operationalisation editor allows the user to build up, and to change, operationalisations written in OF. As with the function net editor, this editor runs under the control of MODPLAN's interactive system. The user can change both the processing functions and the expression for a given operationalisation.

The following operationalisation editor options are available to the user:

Option	Description
Change information processing functions	The user can add a new function to the list of information processing functions, remove a function from this list, or alter one of these functions.
Change assessment expression	The user is shown the original expression, and is asked to enter a new one.
Print operationalisation	The system formats the list of information processing functions, prints them out, and then prints out the assessment expression.
Function net editor	Enters the function net editor.
Run operationalisation	This option causes the function net interpreter to extract the information needed for the operationalisation, and then the the operationalisation is run on the current decision list. The results are printed out as a list of decisions with the figures of merit found by the operationalisation attached to each one.
Save operationalisation	This updates the operationalisation definition whose operationalisation is being edited, by replacing the original information filtering nets and operationalisation with the new ones resulting from editing. The user is returned to the policy menu. This option is identical to the "save nets" option available from the function net editor.

5.3.4. How Policies are Stored by MODPLAN

MODPLAN stores all of it's information about policies as two assertions in the PROLOG database:

(1) Policy definitions. These are stored in a list, containing one entry for each policy the user defines. Each entry is comprised of the policy name, a description of the policy in English, and a list defining how the policy is to be applied at each choice type (ie. it's operationalisations).

(2) Information filtering nets. For each choice type, the procedures for extracting the information for the operationalisations of the various policies are "merged". These procedures take the form of nets of functions, and are merged in order to avoid extracting the same piece of information more than once. The merged nets are stored in a list.

These two structures are, together, called a **policy operationalisation suite**, or "suite" for short. A suite can be stored in a UNIX file, and the user may have many suites available, to be used on different occasions. A sample suite is given in appendix D.

5.4. Learning Choice Making Algorithms

5.4.1. Introduction

Policies and their operationalisations provide the system with one source of automatic choice making algorithms. The other proposed source is heuristics that are learned by the system, from examples presented to it by the user.

How we go about training depends on the nature of the learning algorithm. Some algorithms are able to learn incrementally, ie. they can learn partial heuristics, and use future examples to refine them (in particular, programs which use a version space or version-space-like learning algorithm). There are also learning programs that are only able to learn if all of their training examples are presented en bloc - eg. the CLS/ID3 family of inductive learning programs, and all of the model driven learning programs. In the planning domain, it is felt that an incrementally trainable learning algorithm would be superior. The following should, to some extent, justify this.

The learning system envisaged would work as follows. Depending on the learning algorithm used, it would maintain either a training instance set or an incrementally modifiable rule, for each choice type. The user would be allowed to select one or more choice points from the choice tree as examples for the system. These would be sorted according to choice type.

Since all decisions, and their statuses (untried, succeeded or failed), are stored at each choice point, the learning system can obtain positive and negative training instances from succeeded and failed decisions respectively. An attribute set would be obtained from an information filtering function net previously defined by the user. The training instances would be used to update either the training instance set or the rule (depending on the learning algorithm) for the example's choice type.

This process would be repeated for all of the example choices presented. If the learning system used was able to incrementally update it's rules, processing all of the presented examples would immediately produce appropriate rules for each choice type. If the learning system is inductive, then inductions will have to be performed over each of the accumulated sets of training instances for individual choice types.

5.4.2. Obtaining Example Sets

In obtaining example choices from the choice-point tree, it is desirable that the user should have complete control over which ones are selected. It should be possible for the user to to be able to train the program from single examples of choices made. We also need to be able to use traces. A trace is a sequence of choices connecting two choice-points in the tree. The system would use each choice in the trace as an example. Systems such as Pat Langley's SAGE ([LANGLEY, 1982]) use an "ideal trace", the trace from the starting choice to a successful concluding choice, to supply examples. There is no need, however, to be so restrictive about the traces used by MODPLAN. Even if a trace is part of an aborted branch in the choice-point tree, it can still yield useful information.

It is envisaged that the user would use the refocusing system (see chapter 4) to find a choice or succession of choices to be used as training instances. In the following, a single choice will be treated as a trace with only one example in it. The system is forced to plan up to the first choice in the trace. Then, for each choice point on the trace, the learning system classifies each of the possible decisions as either positive or negative training instances, and uses them to update it's current hypothesis. The plan state is updated each time the system moves from one choice point to the next, and is used as the source of the attributes needed by the learning system.

The main respect in which this differs from a depth-first search generated "ideal trace" (as in SAGE) is that we don't generate mini-search trees for each decision at each choice point examined. What is more, at any given choice, there may be several decisions that are equally good. SAGE has heuristics in it's initial rule set that force it to consider only one path through the solution space. Since there will probably be many possible paths through the solution space that lead to the successful generation of a plan, this would not be acceptable in MODPLAN. MODPLAN's learning program would probably have be able to accept rules with identical condition parts, but different action parts, ie. disjunct concepts, to be able to cope with this.

One of the problems of the method described for acquiring new examples is that we cannot guarantee complete noise immunity. Certainly, there will not be any data errors, since data is extracted directly from the plan. But unless our example sequence of choices is taken from a path that led to a correct plan (possibly optimally, by some arbitrary criterion), we cannot be sure that the positive training instances have been *correctly* classified as positive. It is quite conceivable that, if we continued planning from the chosen sequence, a failure would be encountered, which would propagate back through the sequence. Thus, unless the examples were specially selected by the user so that they never contained false positive classification instances, the learning system would need some degree of immunity to noisy examples.

5.4.3. Possible Learning Algorithms

In deciding what sort of learning program to adopt for the learning of heuristics, the following factors will have to be taken into account:

(1) **Representation** It is assumed that the examples and rules can be conveniently expressed in some formalism. Is it possible that examples and rules could be expressed in the *same* formalism? If so, then we could use a version space type learning program, which would be able to incrementally update it's rules.

(2) **Acquisition of Examples** As far as the use of examples to modify rules is concerned, we can classify learning programs into two groups. The first requires that all examples are presented at once, and a rule produced

to explain all of them. The second can modify it's hypothesis incrementally as new examples are presented - the hypothesis is maintained as an incomplete rule, until sufficient examples have been presented to narrow it down to just one rule. In theory, we could use either type of system for the learning of heuristics. If we adopted the first, however, we would be forced to store all of the examples that have led to the current rule, so that if any new examples are encountered, they can be added to the example set, and a new rule generated. Because we are likely to be using large attribute sets in our examples, this could mean that we use a lot of memory just for storing examples, and that the generation of new rules could be rather time consuming.

(3) **Generality** How general should the learning program be? Should it be an all-singing, all-dancing program, capable of generalising hypotheses, specialising hypotheses, modifying (incorrect) hypotheses, adding hypotheses, reordering hypotheses, etc., or do the limitations of the planning domain mean that we can impose (desirable) constraints on how much it needs to do? We will definitely need to be able to generalise and specialise hypotheses. A limited ability to add hypotheses would probably also find it's way into the system, since we will want to be able to create disjunctive concepts. A disjunctive concept could be represented by a number of rules with identical action parts, and different condition parts. Given that we are allowing disjunctive concepts, some way of reordering them if control errors occur may well be needed.

CHAPTER 6

Results

6.1. Introduction

Most of this paper has so far been concerned with the operation of MOD-PLAN, and the facilities that it makes available to the user. This chapter describes the run-time performance monitoring system, and the results of some of the test runs done with a blocks world example. Information about plan performance can be obtained by the user from the performance assessment functions provided. These assess performance in terms of how well the system fulfilled the goals of the policies listed in 5.2.2.

Three sets of experiments are described. In the first, the choice making algorithms in the two policy operationalisation suites examined were completely unintelligent. No knowledge other than the order of decisions in a list of decisions is used in making selections.

In the second set of experiments, attempts were made to find out how path length in solution space, failure frequency etc., depend on the way in which "ordering" class choices are made. These are choices that do not directly affect the plan, but do affect the *order* in which changes are made to the plan. If these choices are made well, the efficiency of the planner could be improved. For instance, at 'chooseint' choice-points, the system could choose interactions for correction such that in correcting them, other interactions in the list of outstanding interactions are also corrected, as a side-effect.

The final set of tests was an attempt to show that a user could incrementally improve on some initial policy operationalisation suite, using a set of standard choice making functions, and guided by the assessment information available at the end of a planning run. Studying the way in which a *human* makes these improvements would be the first step in *automating* the process.

6.2. The Run-Time Performance Monitoring System

6.2.1. Overview

The run-time performance monitoring system is intended to measure the performance of the planner in terms of how well it achieves the aims of the policies listed in chapter 5; no claims are made for the generality of the measures described here.

Four performance measuring functions are provided. The results obtained by executing one of these functions can be split into two parts - the global measures, which can be taken to measure how well the policies are performing as a whole, and the local measures, which give some kind of assessment of how well the operationalisations at each choice type are performing. Not all of the functions produce local performance measures, but they all produce some global measures. Results are printed out once found.

The following list gives a brief description of each of the performance measuring functions, along with the format they use in displaying their results:

(1) Find a rough measure of the "linearity" of the current plan network. Only global information is printed out.

(2) Find the total length of the path through the solution space, from the initiation of planning to the current choice-point, plus various other measures relating to the way in which individual choice types affect path length. Various pieces of global information are printed, plus a table, giving local information about choice type separation.

(3) Find information on how many of the choices failed. A couple of pieces of global information are printed out, and a table giving local failure frequency, urgency etc. is also displayed.

(4) Find out how much redundancy there is in the final plan. Only global redundancy measures are displayed.

The user is able to make MODPLAN assess progress so far at any choice-point, via the interactive system. As well as being able to perform the above four tests separately, the system can also be made to produce an assessment file, containing the results of all these tests (in the order in which they appear in the above list), for future reference. The experimental results that appear later were obtained in this way.

The following four sections give a more detailed discussion of how the performance assessment functions work.

6.2.2. Assessing Plan Network Linearity

The **linearity** of a plan is a measure of how serial the nodes in the plan network are. In a completely linear plan, all nodes would be in series. A completely parallel plan would have all of it's nodes, apart from the first and the last, in parallel. For a network with no redundant links, and a given number of nodes (N), then, these form the two extremes of linearity and non-linearity. The number of links in the net will be a minimum for a completely linear network, and can be calculated using the formula:

$$L_{min}=N-1$$

The number of links is at a maximum for a completely parallel network, and can be calculated thus (for a network containing more than two nodes):

$$L_{max}=(N-2)*2$$

If the actual number of links is L, which will lie somewhere between L_{min} and L_{max}, we can calculate a "linearity factor", F, which is a comparison between the actual number of links in the network and the two possible extremes:

$$F = \frac{(L-L_{min})}{(L_{max}-L_{min})} = \frac{(L-N+1)}{(N-3)}$$

Since one of MODPLAN's policies is currently "minimise plan network linearity", we would hope that L was as close to L_{max} as possible; in the case of L and L_{max} being equal, F will take on it's maximum value of unity. In the other extreme, if $L=L_{min}$, F will become zero. Thus, the more nearly the network approaches the "ideal linearity", the more closely will F approach unity.

The function which assesses the plan network's linearity will first remove all redundant links from the network. It will count the number of nodes in the network to obtain N, and count up the total number of successors of these nodes; this will give the number of links, L. F is found from these using the above formula, and printed out.

The function that removes redundant links from the network is also used whenever the user asks for a print out of the network, and works as follows. It examines the successor list of each node in the network. If there is only one successor, it is assumed that this cannot be redundant. If there is more than one, each is tested in turn, using a simple algorithm thus:

> *Trace forwards through the network from the node pointed to as a successor. Check each node encountered to see if it is also pointed to by one of the other entries in the successor list. If any nodes are pointed to in this way, remove the link implied.*

The reasoning here is that, if we have a chain of nodes in the network, with two or more having links from a node off the chain attached to them, then only the link that is first, chronologically, is necessary. The system has no a priori knowledge about which link is chronologically first, so for each one, it searches blindly forwards, to see if it can find any of the other links; it knows for sure that any it does find are redundant, since they are linked after the one under test. Subsequent checking may, of course, reveal that the link found to be first by the above test is itself redundant.

6.2.3. Path Length Assessment Techniques

The system is not able to directly compare path length information for plans generated under different policy operationalisation suites, but the information it does produce can be compared manually. The path length assessment function looks only at choices on the path from the starting choice to the current choice-point. The numbers that identify these choices are stored in sequence in a list called the trail.

The only global information to be produced by the path length assessment function is the length of the path up to the current choice, which it finds by counting the number of choices in the trail.

For each choice type, the function finds the following local information:

(1) Number of occurrences. This is a count of the number of times a choice of this type appears on the trail.

(2) Distance to next 'start'. This is the mean distance from a choice of this type to the next choice of type 'start'. This gives a very rough idea of how many choices this choice type is responsible for introducing into the tree. This is not a good means of comparing how good the choice making algorithms for different choice types are, since, under the current control structure, some choice types (eg. 'choosexp') are always a long way from the next 'start', whilst some are always near to it (eg. 'chooseint').

(3) Distance to nearest choice of the same type. This is the mean distance between choices of the same type. This is a slightly better measure than the above for comparing the numbers of choices consequent on a given choice type. However, the more frequent a choice type, the smaller the mean separation will be.

(4) Nearest same type in next stop/start cycle. The current control structure works by making a 'start' choice, then making a sequence of choices of other types, and then making another 'start' choice, terminating with a 'stop' choice. Thus, it is cyclical in nature. We can find the mean distance between the first occurrence of a given choice type in one cycle, and the first occurrence in a following cycle. This is probably the best measure for comparing the operationalisations for different choice types as far as the number of resulting choices is concerned.

6.2.4. Failure Related Plan Assessment Techniques

One method for measuring the "efficiency" of a planner, or indeed, any search technique, is to find out how much effort was wasted on dead-end paths in the solution space. Each choice-point in MODPLAN's tree contains a list of all the possible decisions that could be made at that point, with indications, for those that have been tried already, of whether they succeeded or failed. Associated with each failed decision is a count of the number of choices that were made, and then failed, as a result of that decision. One of MODPLAN's policies is "minimise number of failing choices". By this criterion, a "good" choice making algorithm would select a decision that did not lead to failure, or one that did not lead to very many failures. A "good" suite of policy operationalisations would lead the planner to a correct solution with as few unwanted detours to failing choices as possible.

The failure related plan assessment function produces two global assessments; the "penetrance" and the mean fraction of decisions failing. The penetrance of a search is a measure originally devised by Doran and Michie to measure the performance of their Graph Traverser program ([DORAN, 1966]). It is the ratio of the number of choices on the path leading to success to the total number of choices made during planning. Ideally, this should be unity, indicating that no failures at all had occurred. The other measure used is found as follows. For each choice on the trail, the ratio of the number of failing decisions to the total number of possible decisions is found. The mean value over all of these gives the "mean fraction of decisions failing". This is ideally zero, indicating that no decisions have failed. It is a measure of the probability that a choice will fail under the current suite of policy operationalisations.

There are four local assessments made, for each choice type:

(1) Frequency. This is the ratio of the total number of choices in the trail to the number of times that choices of the given type arise. It is thus a measure of how often choices of this type occur.

(2) Mean fraction of decisions failing. This is similar to the global measure, except the mean is taken only over choices of a given type. This gives a measure of the probability that the operationalisation for this choice type will choose a failing decision.

(3) Extra work weighted mean fraction of decisions failing. This is similar to the above, except that for each choice type, the mean fraction of decisions failing figure is multiplied by a weighting factor related to the amount of extra (unnecessary) work done as the result of selecting "bad" decisions. This factor is calculated by finding the mean number of failing choices resulting from this choice type, and dividing it by twice the mean number of failing choices resulting from *all* choices on the trail. This figure will tend to unity for very bad choice making algorithms.

(4) Urgency. This is simply the ratio of the number of failing choices caused by a given choice type to the total number of failing choices. The nearer it is to unity, the more "urgent" it is that the operationalisation for this choice type should be improved.

6.2.5. Assessing Redundancy

There are four global measures of redundancy produced by the redundancy assessment function, and no local measures. The first two are concerned with contributors to conditions in 'Gost'. They measure the mean number of contributors to conditions in 'Gost', and the condition redundancy factor. The second two are concerned with the 'action' nodes in a plan. They measure the number of action nodes, and the action to goal ratio.

To find the mean number of contributors, the system simply counts up the total number of contributors for all conditions in 'Gost', and divides by the total number of conditions in 'Gost'. The other measure, the "condition redundancy factor" is found by dividing this mean contributor figure by the number of entries in 'Tome'. An indication of how good the system is at creating redundancy, for a given condition, is the ratio of the number of *possible* contributors to the number of contributors *actually* mentioned in it's contributor list. Assuming that the number of possible contributors is roughly proportional to the number of effects logged in 'Tome', the condition redundancy factor should give a fairly plan independent measure of how much redundancy a suite of policy operationalisations introduces. It can be used to compare different policy operationalisation suites.

Without doing a fairly sophisticated analysis of the plan, it is difficult to know which of the actions in it are unnecessary. However, we can make some useful information about actions in the plan available to the user, to make comparisons between different policy operationalisation suites possible. First, we can find the number of actions in the plan. This enables the user to compare how well different policy operationalisation suites tackle a given problem. To give a degree of independence from the actual problem being tackled, the system makes available another measure, the ratio of the number of actions in the current plan to the original number of (user specified) goals.

6.3. Tests and Results

6.3.1. Introduction

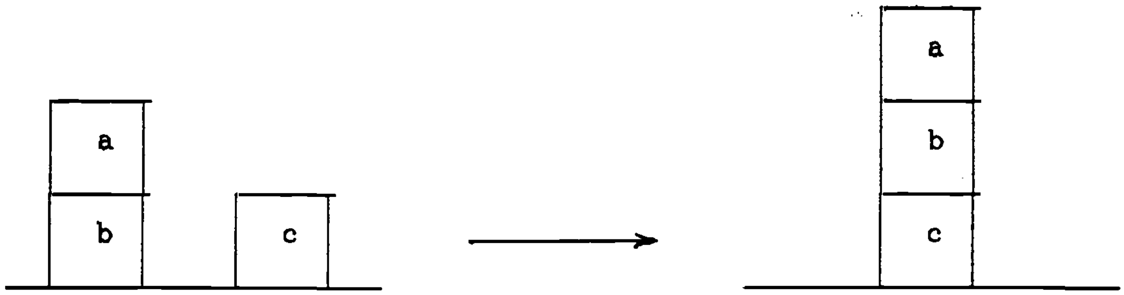
A number of experiments were performed using various policy operationalisation suites on a blocks world example. The example is the same as used by Tate [TATE, 1976]; the initial state of the world is block 'a' on top of block 'b', which in turn is on the table, and block 'c' standing alone on the table. The goals are to achieve 'on(a, b)&on(b, c)', ie. a tower, with 'a' on top of 'b', and 'b' in turn being on top of 'c'.

The following sections give a detailed description of the blocks world problem used, followed by an explanation of the policy operationalisation suites mentioned in the experiments, and then the actual descriptions of the experiments, with their results.

6.3.2. The Three Blocks Problem - It's Representation Under MODPLAN

The blocks world has been used as a domain for testing planning programs for around 20 years, so the reader is probably familiar with it. In dia.6.1, the problem used to test MODPLAN is shown diagramatically. A robot arm must put block 'a' onto block 'b', and put 'b' on top of 'c'.The robot arm cannot carry more than one block at a time, so before moving a block, a test must be made to ensure that it is clear. This means that the order in which the blocks are stacked is important - if the robot puts 'a' onto 'b' before 'b' has been put on 'c', it becomes impossible to put 'b' onto 'c', since 'b' is no longer clear.

The planning operators available to MODPLAN for the blocks world domain are equivalent to those defined by Tate [TATE, 1976]; they are



dia.6.1 The three blocks problem

represented in a Task Formalism that is very similar to the original NONLIN TF, differing only in that a PROLOG syntax is used. A sample of MODPLAN's TF (for the 3-blocks problem) is given in Appendix D.

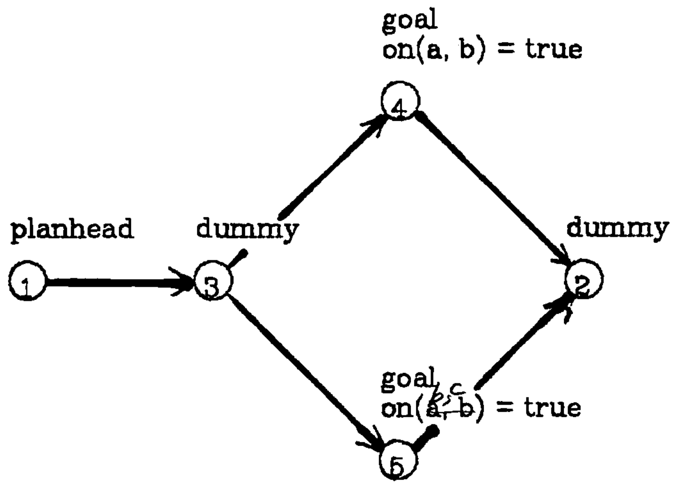
Three operators are defined, two for expanding 'goal' nodes in the plan network, and one for expanding 'action' nodes in the network. 'goal' nodes can have two possible patterns in this domain - either 'on(X, Y)' or 'cleartop(X)'. The two operators for expanding 'goal' nodes were written to cope with these two patterns, inserting appropriate 'goal' and 'action' nodes for putting a block 'X' onto a block 'Y' and clearing a block 'X' respectively. 'action' nodes can only have the pattern 'putontopof(X, Y)' in the blocks world domain. The expansion of such a node does not introduce any new nodes into the plan network, but it will introduce effects, which will be put into 'Tome', and will require conditions to be satisfied, which will be put into 'Gost'.

In order to give some idea of how MODPLAN solves problems, we shall look at the way in which it tackles the 3 blocks problem. The following example was produced by making MODPLAN plan under the policy suite "polics13" (see next section), which enabled MODPLAN to plan without any failures at all.

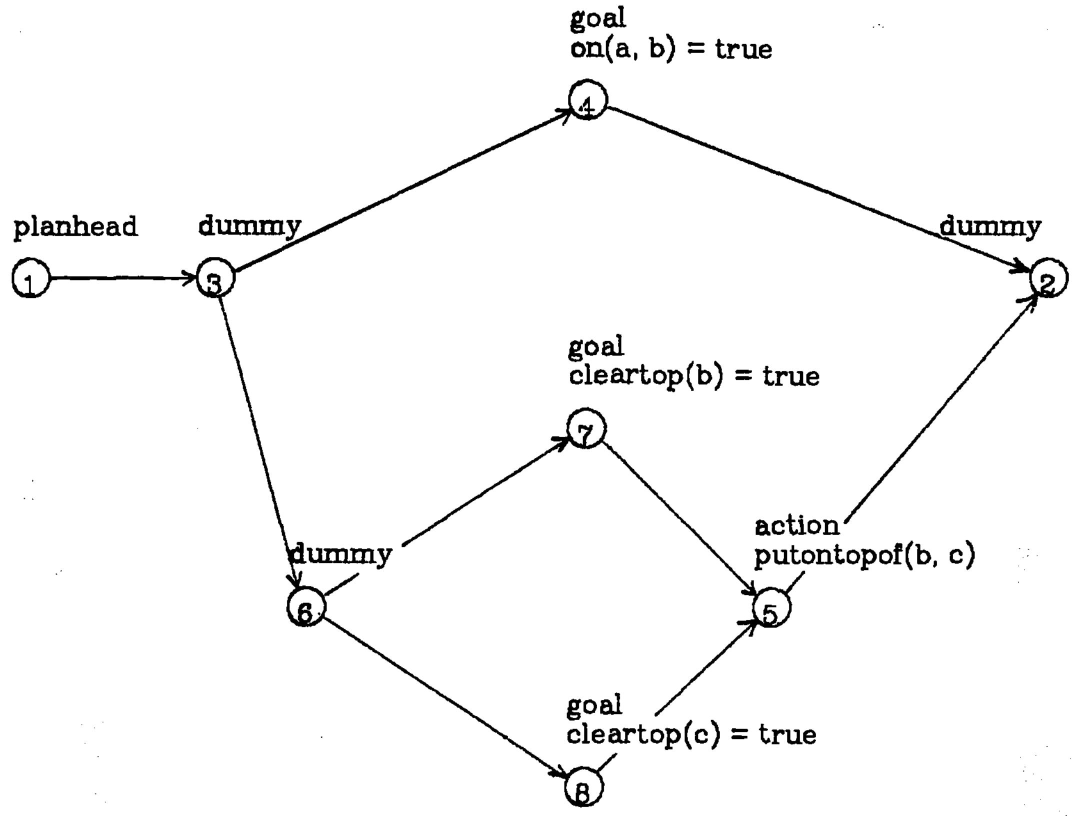
The 'auto' facility used to make the system plan automatically between 'start' choice types. At every 'start' choice, control was passed back to the user, so that a print out of the plan could be obtained. Thus, the changes made to the plan could be examined after each cycle of expansion.

First, MODPLAN found the goals for the problem, that is 'on(a, b)' and 'on(b, c)'. From these, it constructed an initial plan network, in which the two goals were to be solved in parallel, as shown in dia.6.2. Next, node 5 was expanded using the 'makeon' operator, adding the nodes 6, 7 and 8 to the net (see dia.6.3).

In dia.6.4, we see that node 8 has become a 'phantom'. This means that in trying to expand this node, the system has discovered that it is already satisfied. The other thing that is new is that node 7 has become an 'action' node, and a new node, node 9 has been inserted. These are the result of applying the 'makeclear' operator to the original node 7; the system found



dia.6.2 Initial network for three blocks problem

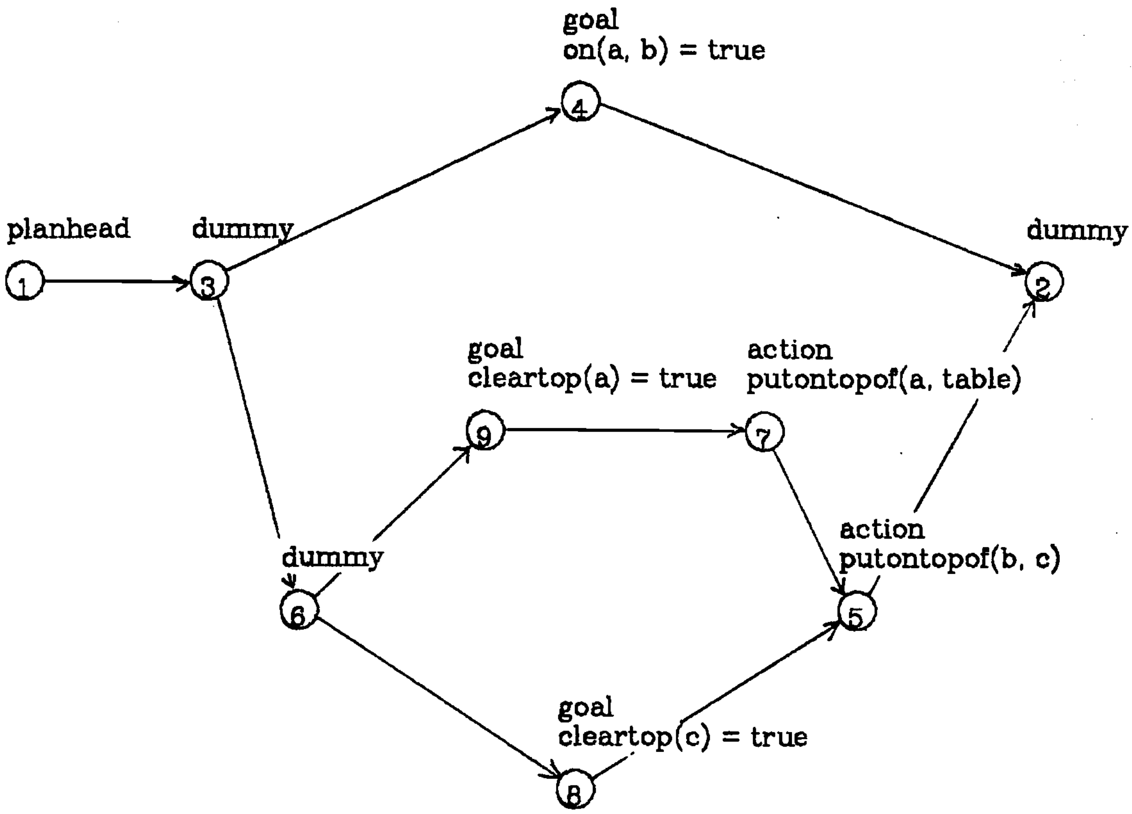


dia.6.3 Plan network after expanding node 5

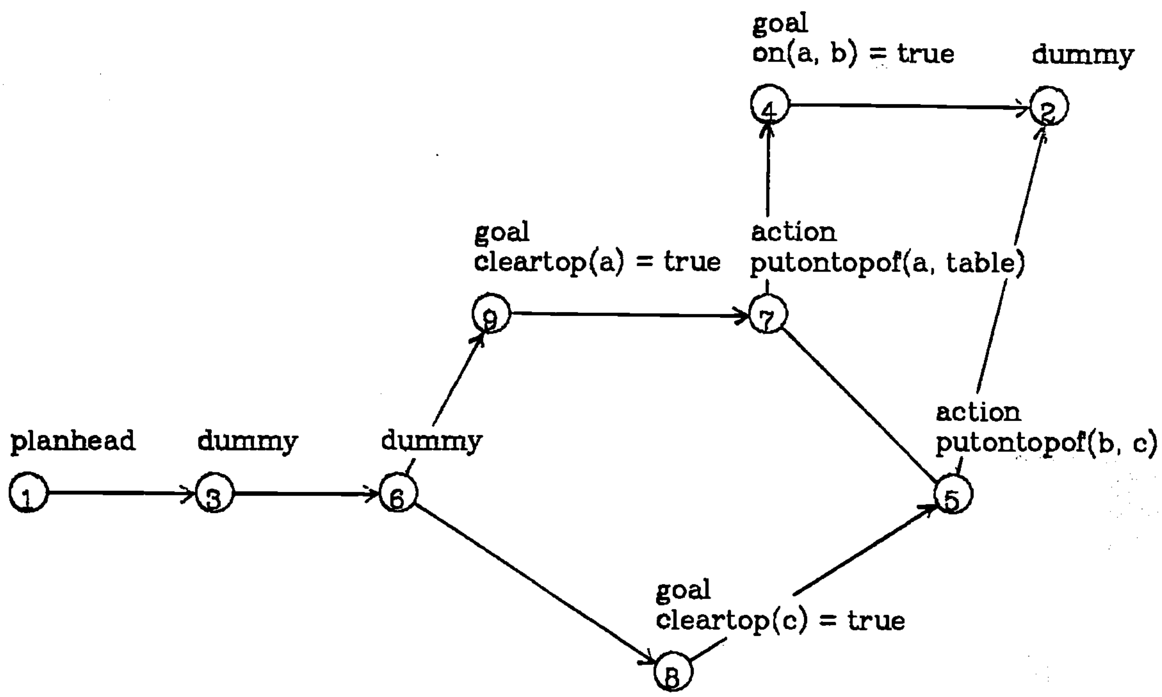
that block 'c' was not clear, therefore it had to be made clear.

dia.6.5 shows the next stage of planning. The 'action' at node 7 had the effect of making 'on(a, b)' false, since it put 'a' onto the table. There was a condition on node 2 saying that 'on(a, b)' must be true, and the only contributor to this condition was node 4. Nodes 4 and 7 were in parallel in the previous plan network, and hence there was an *interaction* between them. To remove this interaction, the system reordered nodes 4 and 7, so that 4 was linked to 7. The link from 3 to 4 became redundant, and was removed.

In the next diagram, dia.6.6, node 4 has been expanded using the 'makeon' operator. All of the actions needed for the complete plan are now in the



dia.6.4 Plan network after expanding node 7

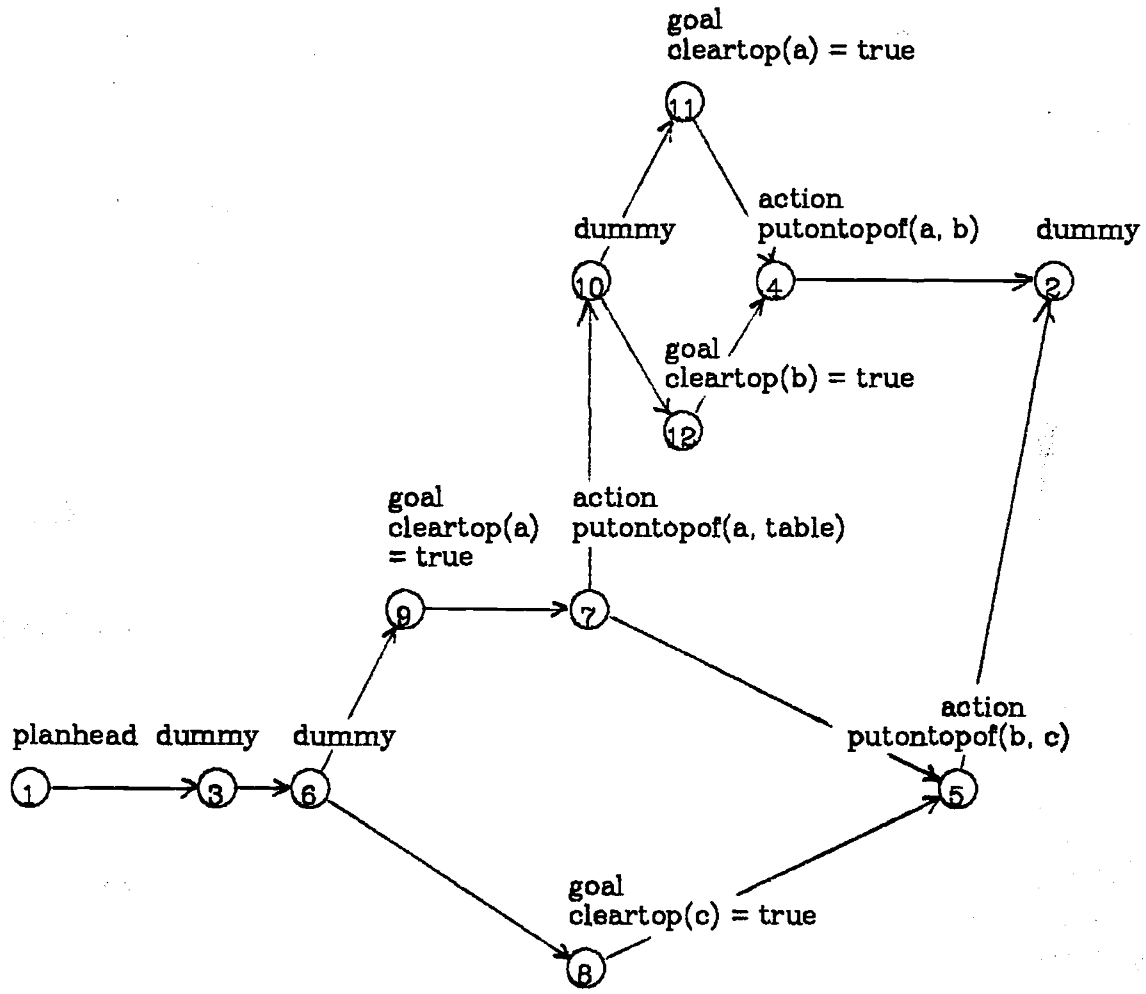


dia.6.5 Plan network after interaction correction

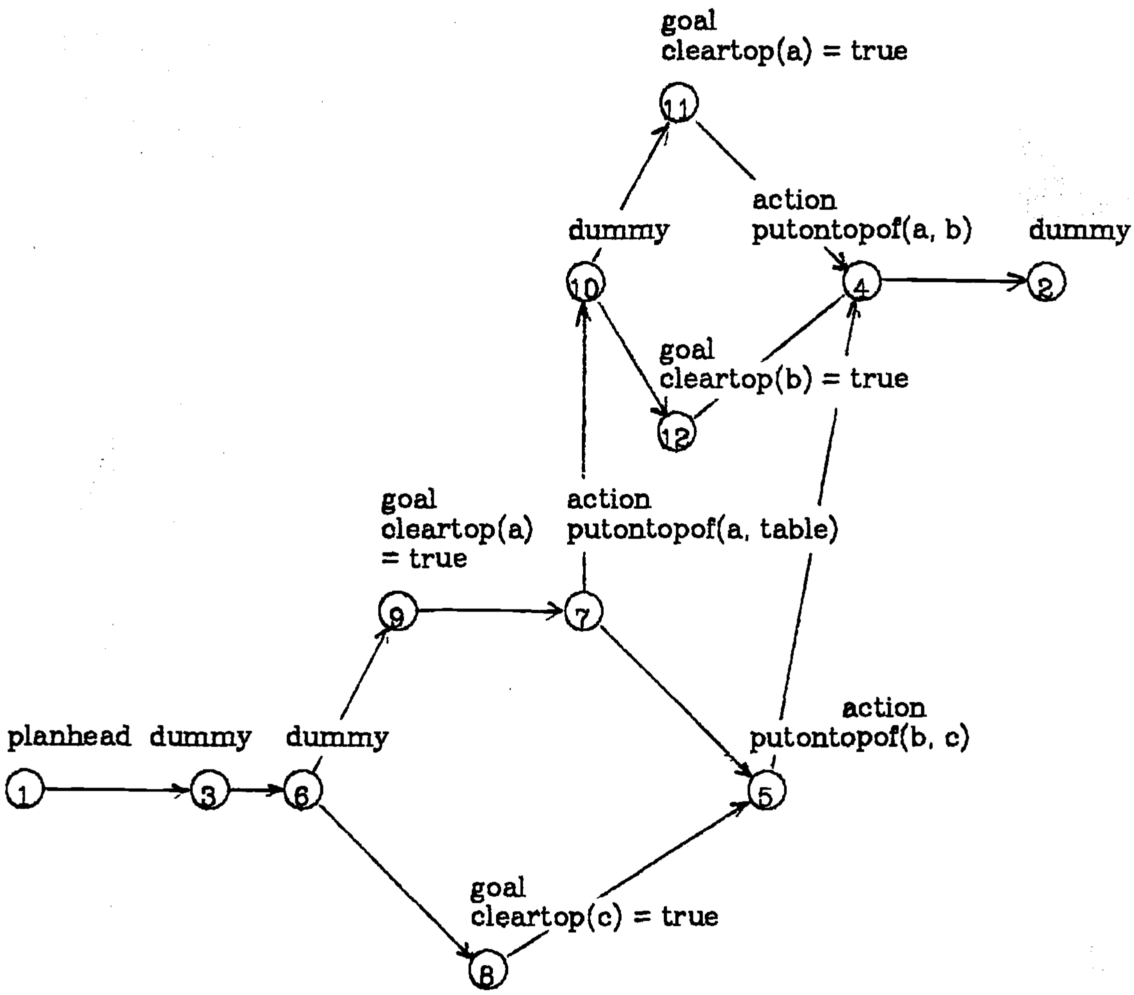
network.

However, the system's troubles were not yet over. Expanding node 4 gave rise to the effect 'cleartop(b)' is false. One of the conditions on node 5 was that 'cleartop(b)' should be true, with a contributor coming from node 7. Thus, we have a second interaction to deal with. The system did this by linking node 5 before node 4; the range over which 'cleartop(b)' needs to be true is now safely isolated from the effects of the action at node 4. This is shown in dia.6.7. We have in fact skipped several cycles ahead of the last network. All of the nodes which were previously 'goal's are now 'phantom's, and this is, in fact the final plan. The sequence of actions is:

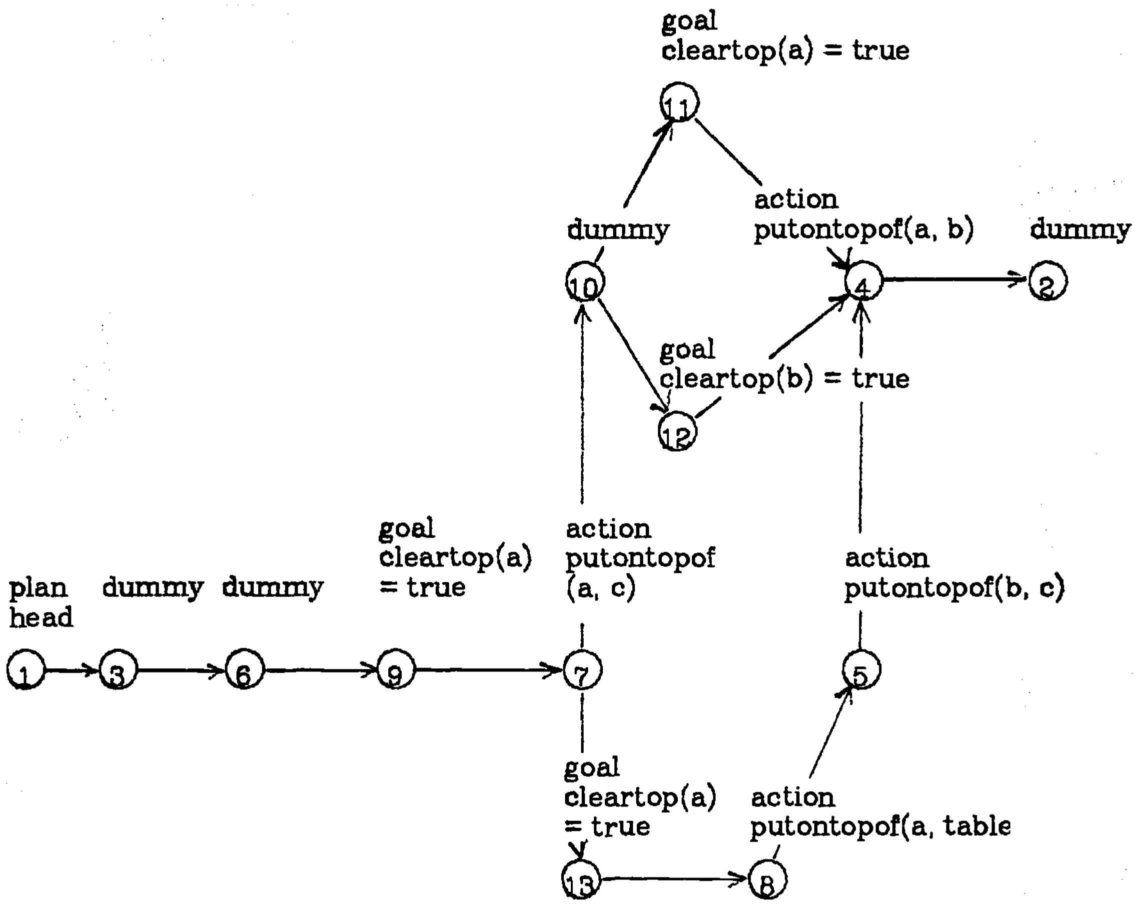
putontopof(a, table); putontopof(b, c); putontopof(a, b).



dia.6.6 Plan network after expanding node 4



dia.6.7 Plan network after second interaction correction



dia.6.8 Non-optimal solution to three blocks problem

6.3.3. Policy Operationalisation Suites Used in the Experiments

A total of 14 different policy operationalisation suites were used in the experiments to be described. A policy operationalisation suite defines, for each policy, the choice making algorithms to be used at all choice types. These suites can be stored in named files; in the experiment descriptions, the names of the files in which suites were stored will be used to identify them. A notional idea of what a policy operationalisation suite looks like is shown in table.6.1.

Policy name	Choice type affected	Choice making function
Minimise redundant actions in plan.	N/A	N/A
Minimise plan network linearity.	N/A	N/A
Find shortest path through solution space.	chooseoper chooseinst chooseint choosecorr choosecontrib choosecond choosesatis	firstdecision firstdecision firstdecision firstdecision firstdecision firstdecision firstdecision
Find a path through solution space not leading to failure.	N/A	N/A
Minimise redundant contributors to a condition.	N/A	N/A
Minimise cost of changes to plan.	choosefact	firstdecision

table.6.1 The "polics2" operationalisation suite

Policies which do not have any choice making functions associated with them are marked by "N/A" in the table. This means, in effect, that these are unimplemented policies, since they do not influence planning in any way.

As mentioned in chapter 5, operationalisations for the experiments are quite simple. In effect, the information filtering function in the function net generates "gradings" for each of the decisions in the decision list. The only thing that needs to be done to the output from this information filtering function is, for each decision in the decision list, divide the corresponding "grading" by 20. This gives assessments of the merit of each decision. There may, of course, be more than one information filtering function in the net. In this case, the system will, for each decision, obtain an assessment from the output of each function, and mediate between them, by obtaining the mean of the assessments found.