Planning For Conjunctive Goals

David Chapman

Abstract

The problem of achieving conjunctive goals has been central to domain-independent planning research; the nonlinear constraint-posting approach has been most successful. Previous planners of this type have been complicated, heuristic, and ill-defined. I have combined and distilled the state of the art into a simple, precise, implemented algorithm (TWEAK) which I have proved correct and complete. I analyze previous work on domain-independent conjunctive planning; in retrospect it becomes clear that all conjunctive planners, linear and nonlinear, work the same way. The efficiency of these planners depends on the traditional add/delete-list representation for actions, which drastically limits their usefulness. I present theorems that suggest that efficient general purpose planning with more expressive action representations is impossible, and suggest ways to avoid this problem.

Copyright © Massachusetts Institute of Technology 1985.

Revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on January 25, 1985 in partial fulfillment of the requirements for the degree of Master of Science.

This report describes research done in part at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grants MCS-7912179 and MCS-8117633, and in part by the IBM Corporation.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, either expressed or implied, of the Department of Defense, of the National Science Foundation, or of the IBM Corporation.

Contents

1 Introduction 1
- 1.1 Nonlinear conjunctive planning . . . 2
- 1.2 Guide to this report . . . 2

2 TWEAK 4
- 2.1 The plan representation . . . 4
- 2.2 Making a plan achieve a goal . . . 10
- 2.3 The top-level control structure . . . 12
- 2.4 Example . . . 14

3 Past and future planning research 18
- 3.1 Chronology . . . 20
- 3.2 Representation . . . 20
  - 3.2.1 Actions . . . 21
  - 3.2.2 Time . . . 25
  - 3.2.3 Codesignation constraints . . . 26
  - 3.2.4 Other problems for future research . . . 27
- 3.3 Making a plan achieve a goal . . . 28
- 3.4 The top-level control structure . . . 33

4 Conclusions 37

A Proofs 38
- A.1 The modal truth criterion . . . 38
- A.2 The outcomes lemma . . . 43
- A.3 The correctness/completeness theorem . . . 44
- A.4 First undecidability theorem . . . 44
- A.5 The intractability theorem . . . 45
- A.6 Second undecidability theorem . . . 47

B Code 48

Plan to capture baby Roo

by Rabbit

1. *General Remarks.* Kanga runs faster than any of Us, even Me.
2. *More General Remarks.* Kanga never takes her eye off Baby Roo, except when he's safely buttoned up in her pocket.
3. *Therefore.* If we are to capture Baby Roo, we must get a Long Start, because Kanga runs faster than any of Us, even me. (*See* 1.)
4. *A Thought.* If Roo had jumped out of Kanga's pocket and Piglet had jumped in, Kanga wouldn't know the difference, because Piglet is a Very Small Animal.
5. Like Roo.
6. But Kanga would have to be looking the other way first, so as not to see Piglet jumping in.
7. See 2.
8. *Another Thought.* But if Pooh was talking to her very excitedly, she might look the other way for a moment.
9. And then I could run away with Roo.
10. Quickly.
11. *And Kanga wouldn't discover the difference until Afterwards.*

— *Winnie The Pooh*

Acknowledgements

My intellectual debt to the great lineage of AI planning researchers is enormous and obvious.

Part of the research described here was done during the summer of 1985 while visiting the SRI AI Center and supported by the Center for the Study of Language and Information. The opportunity provided me by Mike Georgeff and Stan Rosenschein to work with the largest concentration of planning researchers in the world was invaluable.

This report incorporates suggestions from many readers. They made me reformulate TWEAK over and over again. Phil Agre, Steve Bagley, John Batali, Alan Bawden, Mike Brady, Randy Davis, Tom Dean, Gary Drescher, Margaret Fleck, Walter Hamscher, Leslie Kaelbling, Amy Lansky, Scott Layson, Tomas Lozano-Perez, David McAllester, Kent Pitman, Charles Rich, Stan Rosenschein, Mark Shirley, Yoav Shoham, Reid Simmons, Tom Trobaugh, Dan Weld, and David Wilkins contributed much.

Ken Forbus convinced me that my understanding of nonlinear planning would make a Master's thesis. Ed Giniger taught me biology and kept me sane trading stories about idiotic lab politics. My office mate David McAllester was a source of much mathematical wizardry and put up with my randomness. Jim Vanesse and Naomi Leavitt got me through hard times.

My supervisor Chuck Rich supported me through six thesis topic changes and believed in me when I didn't. His ability to debug me when wedged was vital.

Chapter 1

Introduction

If you intend to use a domain-independent planner as a workhorse black-box part of something else, you care whether it works. Planners of the most promising ("nonlinear") sort have been complicated, heuristic, ill-defined AI programs, without clear conditions under which they work. This report describes a nonlinear planner, TWEAK, that has few novel features, but is a simple, precise algorithm I have proved correct and complete.

I started work on planning because I wanted a planner to co-routine with a learner to make an integrated problem-solver [4]. I'd heard that Sacerdoti's NOAH was the state of the art in planning, and decided to copy it exactly, since I had no interest in the matter. Four readings of [44] and three misconceived implementations later, I had a planner that worked, but no idea why. To determine whether it would work as a reliable subroutine, I had to simplify the algorithm and representations and apply some mathematical rigor. To quote Sacerdoti:

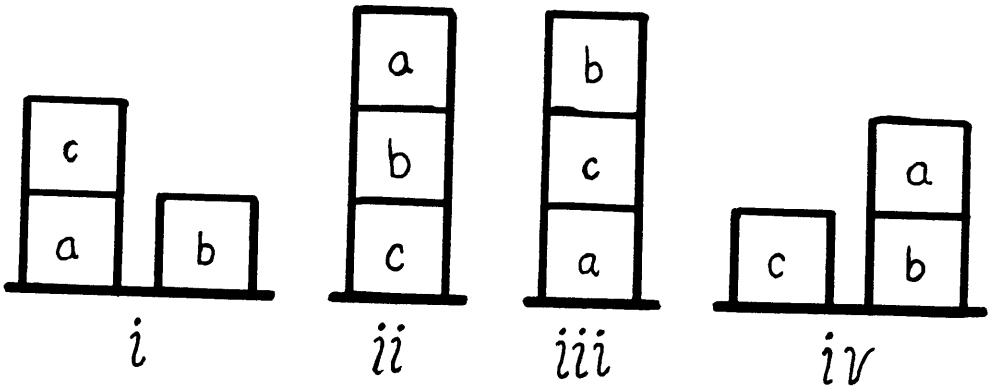
> [The basic operations of NOAH] were developed in an ad hoc fashion. No attempt has been made to justify the transformations that they perform, or to enable them to generate all transformations. However, it should be possible to define an algebra of plan transformations ... a body of formal theory about the ways in which interacting subgoals can be dealt with.[45]

That is what I've done in this report.

Rigor of formulation not only gives confidence in a program, it may be needed as a stable base for further research. AI comes in "neat" and "scruffy" styles. Planning research to date has been mainly scruffy: heuristic, ill-understood, unclear. As I discovered, scruffy research is hard to duplicate. That is not Sacerdoti's fault, or mine: most AI research is necessarily like that. When working at the frontiers of knowledge, you will make no progress if you wait to proceed until you understand clearly what you are doing. But it is also hard to know when progress has been made and where to go next before the scruffy work is neatened up. Neat and scruffy research on a particular domain should follow each other in cycles. Late in this neat report, I will make some scruffy suggestions about how to go beyond the crucial limitation of the domain-independent planners that have been implemented to date.

1.1 Nonlinear conjunctive planning

The conjunctive planning problem has been a main focus of planning research for the last ten years. The problem is to achieve several goals simultaneously: to find a plan that makes a conjunctive formula true after it has been executed. To make a planner generally useful, it should be domain-independent. The difficulty in domain-independent conjunctive planning is in interactions between the means of achieving the individual goals. The following classic problem, known as the "Sussman anomaly", illustrates the difficulty. Suppose we have three blocks, a, b, and c; initially c is on a and a and b are on the table (situation *i* in the figure). We want to have a stacked on b on c, or to achieve the conjunctive goal (and (on a b) (on b c)) (situation *ii*). Let's say you're only allowed to move one block at a time, so that the top of a block must be clear before it can be moved. If you try to put b on c first, when you go to put a on b you fail, because c is on a and so prevents it from moving (situation *iii*). On the other hand, if you try first to put a on b (removing c to make a accessible), putting b on c is made impossible by a, which is in the way (situation *iv*).



I'll return to this problem later in the report and show how nonlinear planning can solve it. The important idea, due to Sacerdoti, is that a plan (at least while it is being constructed) does not have to specify fully the order of execution of its steps. In other words, a plan is only a partial order on steps; this is what is meant by *nonlinear* planning.

1.2 Guide to this report

The next chapter explains how and why TWEAK works. The chapter is divided into three sections: the first explains what a plan is; the second shows how to improve incomplete plans; and the third describes the overall control structure of the planner.

Chapter three covers related and future work. I analyze previous planning research using the analytical tools developed in chapter two, showing that all domain-independent conjunctive planners work the same way. I suggest that the restrictions on representations of actions that these planners depend upon are their crucial limitation, and show that there are complexity-theoretic barriers to lifting these restrictions.

The last chapter presents brief conclusions.

An index appears at the end; this may be useful in keeping track of the many definitions.

Chapter 2

TWEAK

TWEAK is a rigorous mathematical reconstruction of previous nonlinear planners. TWEAK is also an implemented, running program. This chapter describes the algorithm and proves it correct. TWEAK comes in three layers: a plan representation, a way to make a plan achieve a goal, and a top-level control structure. Each layer is described in more detail in one of the next three sections of this chapter. The fourth section gives an detailed scenario of TWEAK solving the Sussman anomaly problem described in the last chapter.

The plan representation is the most complex layer. The basic operation provided by this representation determines whether a proposition will be true of the world after part of a plan has been executed. An efficient algorithm for this operation depends on a subtle theorem about incompletely defined plans, given in section 2.1. Section 2.2 describes a nondeterministic procedure that transforms a plan so that it achieves a goal that it previously did not. The top-level control structure, described in 2.3 controls this nondeterminism. Because choosing how to make a plan achieve a goal is difficult, backtracking search is used to recover from wrong choices. Here I prove that TWEAK is complete: if a solution to a problem exists, TWEAK will find it.

2.1 The plan representation

In this section I define plans, problems, and what it means for a plan to solve a problem. I present a criterion which allows TWEAK to reason about what is true in the world as a plan is executed. Inevitably most of this section is composed of dry and obvious definitions. The index may be useful in keeping track of these. The proofs are deferred to appendix A.

TWEAK is a *constraint posting* planner. Constraint posting is the process of defining an object, a plan in this case, by incrementally specifying partial descriptions it must fit. Alternatively, constraint posting can be viewed as a search strategy in which, rather than generating and testing specific alternatives, chunks of the search space are progressively removed from consideration by constraints that rule them out, until finally

every remaining alternative is satisfactory. The advantage of the constraint posting approach is that properties of the object being searched for do not have to be chosen until a reasoned decision can be made. This reduction of arbitrary choice often reduces backtracking.

As TWEAK works on a problem, it has at all times an *incomplete plan*, which is a partial specification of a plan that may solve the problem. This incomplete plan could be completed in many different ways, depending on what constraints are added to it; thus it represents a class of complete plans. The incomplete plan supplies partial knowledge of the complete plan that will eventually be chosen; planning is finished when all the completions of the incomplete plan solve the given problem. I will say "*necessarily p*" if p is true of all completions of an incomplete plan, and "*possibly p*" if p is true of some completion. The number of completions of a plan is exponential in its size, so computing whether something is possible or necessary by searching completions would be very expensive. The heart of this section is a polynomial-time algorithm that computes possible and necessary properties of an incomplete plan.

TWEAK's plan representation is very simple. In fact, it is so restrictive that it is impossible even to represent most domains; I will explain why in section 3.2.1. A *complete plan* is a total order on a finite set of *steps*. The order represents time; the steps, actions. The plan is executed by performing the actions corresponding to the steps in the order given. A step has a finite set of *preconditions*, which are things that must be true about the world before the action can be performed. A step also has finitely many *postconditions*, which are things that the corresponding action guarantees will be true about the world after it has been performed. Pre- and postconditions are both expressed as *propositions*. Propositions have a *content*, which is a tuple of *elements*, and can be *negated* or not. Elements can be *variables* or *constants*; there are infinitely many of each. Functions, propositional operators and quantification are not allowed: all propositions are function-free atomic. (See section 3.2.3 to understand why there must be infinitely many variables and constants, and section 3.2.1 for why propositions must be so simple.) Two propositions are *negations* of each other if one is negated and the other is not and they have the same content (strictly, necessarily codesignating content, a notion that I haven't introduced yet). I'll write propositions like this: (on a x) and this: ~(on a x). These two propositions have the same content tuple, the three elements on, a, and x; the second is negated and the first is not.

Plans in TWEAK can be incomplete in two ways: the time order may be incompletely specified, using *temporal constraints*, and steps may be incompletely specified, using *codesignation constraints*. A temporal constraint is a requirement that one step be before another; thus a set of temporal constraints is simply a partial order on steps. A *completion* of a set of temporal constraints C is any total order O on the same set of steps such that sCt implies sOt. (Every ordering in the incomplete plan must also hold in the complete one).

Codesignation is an equivalence relation on variables and constants. In a complete plan, each variable that appears in a pre- or postcondition must be constrained to codes-

ignate with (effectively, be bound to) a specific constant. In execution, that constant will be substituted for the variable when the action is performed. Codesignation constraints enforce codesignation or noncodesignation of elements. Distinct constants may not codesignate. Two propositions codesignate if both are negated or both are not and if their contents are of the same length and if corresponding elements in the contents codesignate. For example, the propositions (on a x) and (on a y) codesignate iff x and y codesignate.

Recall the general definitions of necessary and possible; thereby two propositions in an incomplete plan necessarily codesignate if they codesignate in all completions; in other words no matter what constraints are added. You can constrain two not necessarily noncodesignating propositions to necessarily codesignate by constraining all the corresponding elements of their contents to codesignate. This amounts to unification of the two propositions. You can constrain two not necessarily codesignating propositions to necessarily not codesignate by choosing some tuple index and constraining noncodesignation of the two elements at that index in the content tuples of the two propositions. For example, (on a x) and (on a y) can be made to necessarily codesignate by making x and y necessarily codesignate, and to necessarily not codesignate by making x and y necessarily not codesignate.

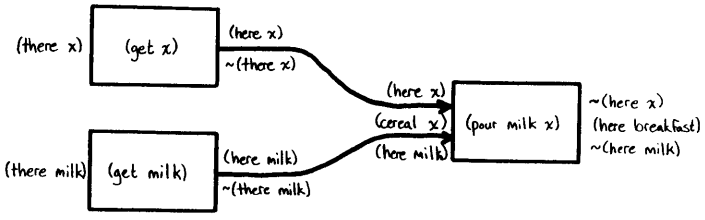
As steps are executed the state of the world changes. TWEAK represents states of the world with *situations*, which are sets of propositions. A plan has an *initial situation*, which is a set of propositions describing the world at the time that the plan is to be executed, and a *final situation*, which describes the state of the world after the whole plan has been executed. Associated with each step in a plan is its *input situation*, which is the set of propositions that are true in the world just before it is executed, and its *output situation*, which is the set of propositions that are true in the world just after it is executed. In a complete plan, the input situation of each step is defined to contain the same set of propositions as the output situation of the previous step. The final situation of a complete plan has the same set of propositions in it as the output situation of the last step. The time order extends to situations: the initial and final situations are before and after every other situation respectively. The input situation of a step is before the step and after every other situation that is before the step; the output situation of a step is after the step and before any other situation that is after the step.

A proposition is *true in a situation* if it codesignates with a proposition that is a member of the situation. A step *asserts* a proposition in its output situation if the proposition codesignates with a postcondition of the step. A proposition is asserted in the initial situation if it true in that situation. A proposition is *denied* in a situation if its negation is asserted there. It's illegal for a proposition to be both asserted and denied in a situation.

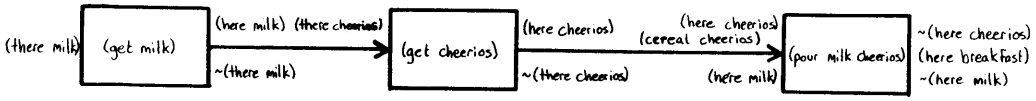
A step can be executed only if all its preconditions are true in its input situation. In this case, the output situation is just the input situation minus any propositions denied by the step, plus any propositions asserted by the step. (The order matters. Also, this is not the same thing as the input situation plus the propositions asserted by the step:

if p were true in input situation and the step asserts $\sim p$, then the output situation must not contain both p and $\sim p$; input and output situations must be consistent sets of propositions, since they describe states of the world.) This model of execution does not allow for indirect or implied effects of actions or for uncertainty of execution; any changes in the world must be explicitly mentioned as postconditions. I will have more to say about this restriction in section 3.2.1.

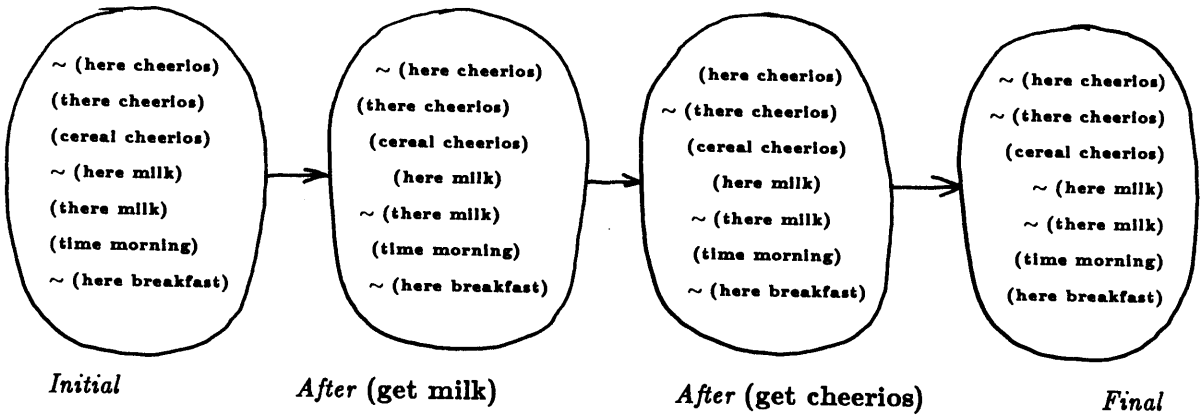
I use graphs, as in the figures below, to illustrate plans. Steps are boxes; the preconditions are put before or above the box and postconditions after or below. The steps may have labels inside, but these are only mnemonic. Arcs represent the partial time order. Ovals are situations.



An incomplete plan.

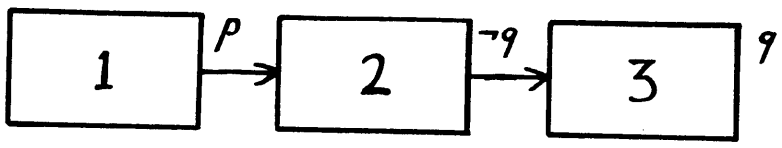


One completion of the example plan.



The sequence of situations resulting from execution of the completion.

During planning, incompleteness introduces uncertainty into the meaning of a plan. To use a blocks world example, if v is a variable, after asserting (on block v), there's no way to tell whether (on block c) is true or false, unless v codesignates with a particular constant. I will now sketch the derivation of a criterion that tells you when a proposition is necessarily true in a situation. Of course a proposition is necessarily true in situation if it is necessarily asserted in it. Once a proposition has been asserted, it remains true until denied. Thus a proposition p is necessarily true in a situation if there is some previous situation in which it is necessarily true, and no possibly intervening step possibly denies it: for if there is a step that is even possibly in-between that even possibly denies p, there is a completion in which the step actually is in-between and actually denies p. (A step possibly denies p by denying a proposition q which possibly codesignates with p). Oddly, the converse of this criterion is not true; this incomplete plan illustrates an exception:

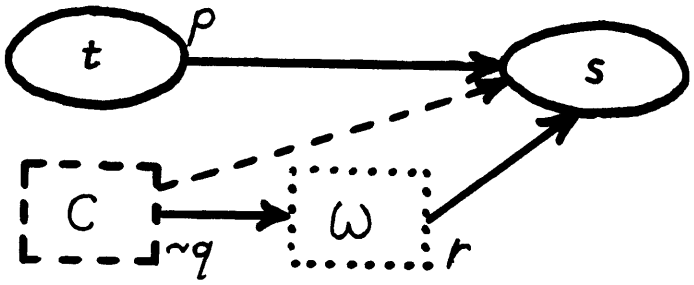


An odd plan

If p and q are possibly codesignating, this plan has two classes of completions: one in which p and q actually codesignate, in which case p is asserted by step 3; and one in which p and q are noncodesignating, so that p is asserted by step 1, and is never denied. In either case, p is true in the final situation, even though no one step necessarily asserts p without an intervening step possibly denying it. The complete criterion, extended to cover such cases, is this:

Modal truth criterion: A proposition p is necessarily true in a situation s iff two conditions hold: there is a situation t equal or necessarily previous to s in which p is

necessarily asserted; and for every step C possibly before s and every proposition q possibly codesignating with p which C denies, there is a step W necessarily between C and s which asserts r, a proposition such that r and p codesignate whenever p and q codesignate. The criterion for possible truth is exactly analogous, with all the modalities switched (read "necessary" for "possible" and vice versa).



The necessary truth criterion. Solid lines indicate necessarily time-relatedness and dashed lines possible time-relatedness; the dashed box, a disallowed step; the dotted box a step that would make the dashed step legal.

I call situations t necessarily before s that necessarily assert p *establishers*; steps C defined as in the statement of the theorem *clobberers*; and steps W that foil steps that would otherwise become clobberers, *white knights*. If a step C is before t, certainly it does not clobber p; but in such a case, the step of which t is the output situation acts itself as a white knight.

The part of the criterion about white knights is counter-intuitive, but it is needed, as illustrated by the previously illustrated odd plan. More complex codesignation implications can also occur: for example, the propositions (x y) and (y z) must codesignate if (x y) and (z x) do.

The truth criterion can usefully be thought of as a completeness/soundness theorem for a version of the situation calculus.

The criterion can be interpreted procedurally in the obvious way (see appendix B). It runs in time polynomial in the number of steps: the body of the criterion can be verified for each of the n^3 triples $< t, C, W >$ with a fixed set of calls on the polynomial-time constraint-maintenance module. (The exponent in this polynomial can be reduced with dynamic programming; this is essentially what Tate's GOST does [63].) However, the modal truth criterion does exponentially much "work" by describing properties of the exponentially large set of completions of an incomplete plan. (Strictly, there may be infinitely many completions, since there are infinitely many constants the variables could codesignate with; but since all but finitely many constants are unconstrained and so equivalent, there are only exponentially many that are not isomorphic.) The remainder of TWEAK depends heavily on the criterion.

Now I will define problems and their solutions. A *problem* is an *initial situation* and a *final situation*, which are two sets of propositions. A *plan for a problem* is one such that every proposition in its initial situation is true in the initial situation of the problem. A *goal* is a proposition which must be achieved (true) in a certain situation. The goals of a plan for a problem are defined to be the propositions in the final situation of the problem, which must be true in the final situation of the plan, and the preconditions of steps in the plan, which must be true in the corresponding input situations. A complete plan for a problem *solves* the problem if all its goals are achieved. Thus, a complete plan solves a problem if it can be executed in the initial situation of the problem and if the final situation of the problem is a correct partial description of the world after execution. The aim of TWEAK is to produce a plan that necessarily solves the problem it is given. This plan may be incomplete, in which case any of its completions can be chosen for execution.

2.2 Making a plan achieve a goal

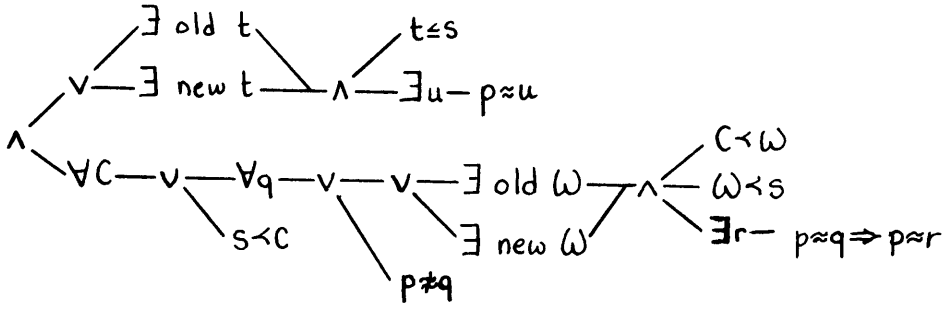
TWEAK's contract is to produce a plan for a specific problem it is given. TWEAK has at all times an incomplete plan, initially null, which is an approximation to a plan that solves the problem. The top-level loop of the planner is to choose a goal and to try to make the plan achieve that goal. This section describes TWEAK's procedure for making the plan achieve the goal.

The *goal-achievement procedure* is derived by interpreting the necessary truth criterion as a nondeterministic procedure. Universal quantification over a set becomes iteration over that set; existential quantification a nondeterministic choice from a set; disjunction a simple nondeterministic choice; and conjunction, several things that must all be done. Also, an existentially quantified situation can be instantiated nondeterministically either by choosing an existing situation in the plan or by adding a new step to the plan and taking its output situation as the value of the existentially quantified variable. To make a situation be before another or to make two propositions codesignate or not codesignate, the procedure just adds constraints. These constraints may be incompatible with existing constraints: for example, you can't constrain s before t if you have already constrained t before s. The constraint maintenance mechanism signals failure in these cases, and the top-level control structure backtracks. Since the set of things of things possibly asserted in a situation can not be changed, to make a proposition necessarily asserted there, the procedure constrains codesignation of the given proposition with one of those asserted.

Let $\approx$ and $\not\approx$ stand for codesignation and noncodesignation respectively and let $\prec$ represent the time order. Then the necessary truth criterion in logical notation reads thusly:

$$\exists t \ \Box\, t \preceq s \ \wedge \ \Box \text{asserted-in}(p,t)$$
$$\wedge$$
$$\forall C \ \Box\, s \not\preceq C \ \vee$$
$$\forall q \ \Box \neg \text{denies}(C,q) \ \vee$$
$$\Box\, q \not\approx p \ \vee$$
$$\exists W \ \Box\, C \prec W \ \wedge$$
$$\Box\, W \prec s \ \wedge$$
$$\exists r \ \text{asserts}(W,r) \wedge p \approx q \Rightarrow p \approx r$$

The following diagram defines the nondeterministic procedure:



This figure is a parse tree of the necessary truth criterion, modified according to the paragraph above on procedural interpretation. In the diagram, $\vee$ means to choose one of the alternate paths; $\wedge$ tells you to do all the paths; $\exists$ means "choose a"; $\forall$ tells you to apply the following path to every one. The leaf nodes are constraints that should be added to the plan. u ranges over the propositions necessarily true in t; q ranges over postconditions of C; r over postconditions of W. Choosing t and W may or may not require the introduction of a new step, as explained earlier.

Code for the nondeterministic procedure appears in appendix B. Alternative paths through this procedure are called *plan modification operations*. The path that leads to constraining $s \prec C$ I call *promotion*.

A number of further comments are needed to fully specify the details of the procedure. Step addition involves choosing what step to add. Every step in a plan must represent an action that is possible to execute in the domain in which the problem is specified. Even to possibly achieve a goal p by addition, the added step must assert a

proposition possibly codesignating with p. The choice of steps, then, is among those that are allowed in the domain and that possibly assert the desired goal. The user must supply TWEAK with a set of template steps that TWEAK can use. These are *instantiated* by copying the step, proposition, variable, and codesignation constraint datastructures. Constants are not copied; if a step template refers to a specific object in the world, all instances should refer to the same one.

Making $p \approx q$ imply $p \approx r$ is tricky; this can not be directly expressed as a constraint. It is sometimes possible to constrain $p \approx q \Rightarrow p \approx r$ with a simpler constraint than either $p \not\approx q$ or $p \approx r$. (w x y) and (u y z) can be made to codesignate in case (w x y) and (w z x) do by constraining w $\approx$ u. In general, there may be several ways to ensure a codesignation implication using constraints between elements, and one of these must be chosen nondeterministically.

One additional way to achieve a goal is imaginable: to remove a clobbering step C. This would not help: every step is introduced to assert some goal proposition, and removing one makes negative progress. Moreover, the search control structure guarantees that the same plan without the clobbering step will be found eventually anyway, and it is never the case that the only way to achieve a goal is to remove a step. Apart from this, the achievement procedure encompasses *all* the ways to make an incomplete plan achieve a goal, because the modal truth criterion is sufficient as well as necessary. So in this respect TWEAK can not be improved upon.

The goal-achievement procedure has the useful property that so long as step addition and removal are avoided, the new plan will continue to necessarily achieve any goals that it previously did. That's because the rest of the procedure operates only by adding constraints. When constraints are added, things that were previously possibly true become either necessarily true or necessarily false, but nothing that is necessarily true can change its truth value.

Step addition adds new preconditions to the plan that need to be achieved, and the added step may also deny, and so undo, previously achieved goals. This is unavoidable, and it can lead to infinite looping. Therefore, TWEAK prefers constraint posting to step addition.

2.3 The top-level control structure

TWEAK begins work on a problem with a first incomplete plan whose initial situation is the initial situation of the problem and which has no steps or constraints. It then enters a loop in which some goal not yet achieved is chosen and the procedure of the last section is applied, yielding a new plan. When all the plan's goals are achieved, the plan necessarily solves the problem. Choosing which goal to achieve and which choices to make in the achievement procedure is very difficult; certainly it is not always possible to choose right the first time. Therefore the top-level control structure of TWEAK is a search through the space of alternate paths through the goal achievement procedure.

People have thought a lot about what sort of search to use; this work is reviewed

in section 3.4. Since none of the search strategies developed so far seem very good, I simply use dependency-directed breadth-first search in TWEAK. I shan't argue for breadth-first search; it's certainly too expensive for general use. However, the use of dependency-directed search deserves some justification.

Dependency-directed backtracking [11] is more efficient than chronological backtracking only if the search space is nearly decomposable into independent subparts, so that after a failure in one part, only the work done on that part needs to be undone; work on other parts can be saved. TWEAK does have this property when running in many domains. For example, in the blocks world, if the goal is to build two disjoint structures, the search space can be divided into the part concerned with building the one and the part concerned with building the other. Failure in building the one structure will not affect partial successes achieved thus far in building the other.

Because step addition can make the plan grow arbitrarily large, the search may never converge on a plan that necessarily solves the problem. In fact, there are three possible outcomes: success, in which a plan is found; failure, when the planner has exhaustively searched the space of sequences of plan modification operations, and every branch fails; and nontermination, when the plan grows larger and larger and more and more operations are applied to it, but it never converges to solve the problem.

Outcomes lemma: Each of the three outcomes is possible for some choice of domain and problem.

This is a central theorem of this report:

Correctness/completeness theorem: If TWEAK, given a problem, terminates claiming a solution, the plan it produces does in fact solve the problem. If TWEAK returns signalling failure or does not halt, no solution exists.

This theorem leaves little room for improvement. Perhaps loop-detection heuristics or techniques for proving no solution exists could make TWEAK loop infinitely less often. An obvious question is whether planning is in fact decidable: whether it is possible to make a complete planner that always halts.

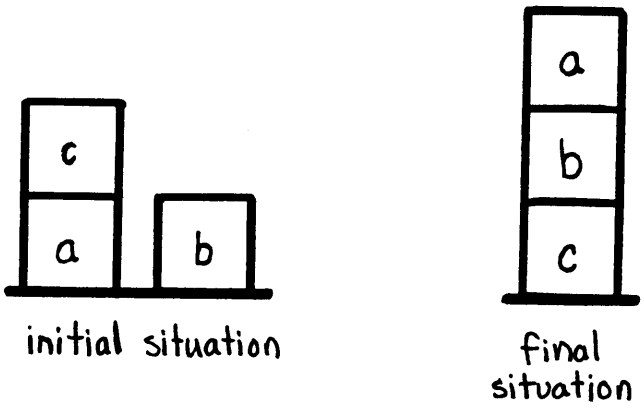
First undecidability theorem: Any Turing machine with its input can be encoded as a planning problem in the TWEAK representation. Therefore, planning is undecidable, and no upper bound can be put on the amount of time required to solve a problem.

This theorem is weaker than it may appear, for two reasons. First, the proof uses an infinite (though recursive) initial state to model the connectivity of the Turing machine's tape. It may be that if problems are restricted to have finite initial states, planning is decidable. (This is not obviously true, though. A finite initial state does not imply a finite search space. There are infinitely many constants, and an action can in effect "gensym" one by referring to a variable in its postconditions that is not mentioned in its preconditions.) Section 3.2.1 shows in passing that planning is undecidable with even

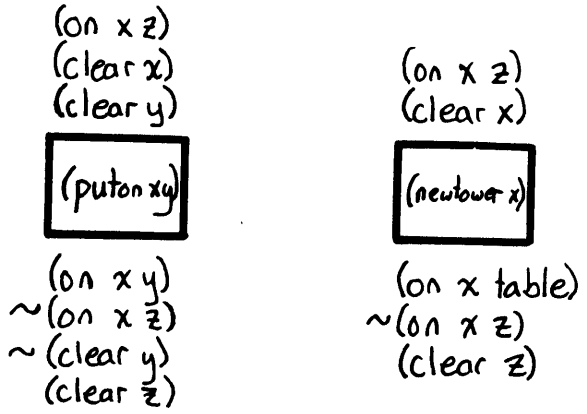
finite initial states if TWEAK's representation is extended a little. Second, what the proof shows is that the size of the shortest plan to solve a problem may be arbitrarily large, rather than that the process of planning itself is complex. In fact, no backtracking is required to solve the Turing machine-encoding problems.

2.4 Example

To give a feel for TWEAK doing its thing, I'll show how it solves the Sussman anomaly problem introduced in section 1.1. Here is the problem again:

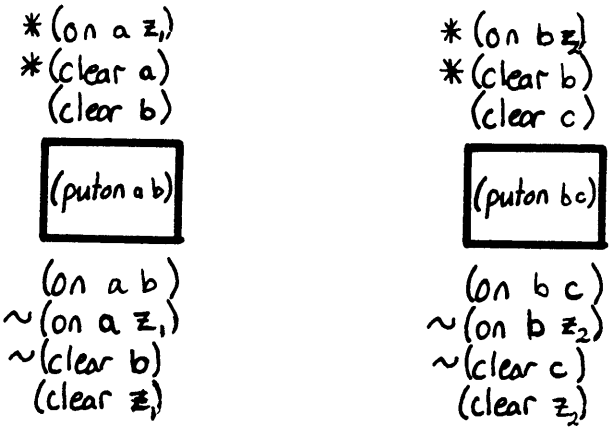


Logically, only one action is available in this domain, puton. puton has three preconditions, (on x z), (clear x), and (clear y); and four postconditions, (on x y), ~(on x z), ~(clear y), and (clear z). z represents the block on which x lies before the puton takes place. There also must be noncodesignation constraints to the effect that x, y, and z are all distinct, and that x is not the table, which can't be moved. This isn't quite good enough: the table is always clear (always can have more put on it), and putting something on the table doesn't change that. We must constrain y not to codesignate with the table and use a different action, newtower, to put a block onto the table (and so start a new tower). newtower has preconditions (on x z) and (clear x) and postconditions (on x table), ~(on x z), and (clear z). Codesignation constraints ensure that x, z, and the table are all distinct.



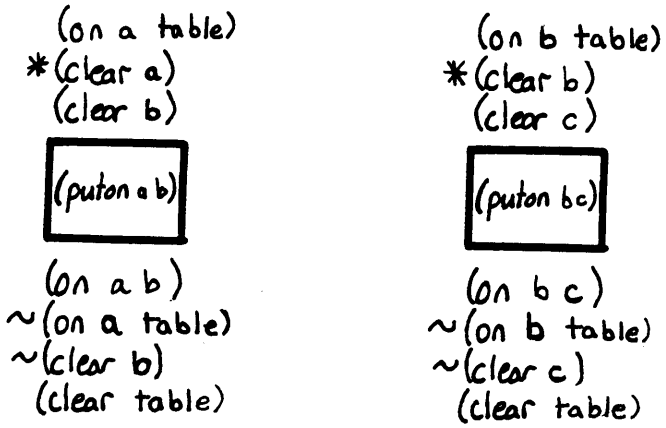
The blocks world step templates.

I'll assume that the nondeterministic control structure always guesses right the first time. TWEAK begins with the two top-level goals (on a b) and (on b c). Each is achieved by adding a puton step:

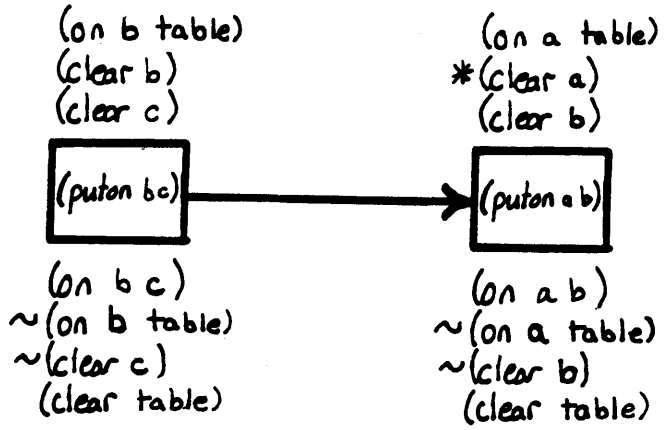


The variables instantiating x and y are constrained to specific values by virtue of the postconditions of the two steps having been constrained to codesignate with the goals. I have been somewhat sloppy and textually substituted constants for the variables that are bound to them. z_1 and z_2 are the instantiations of z.

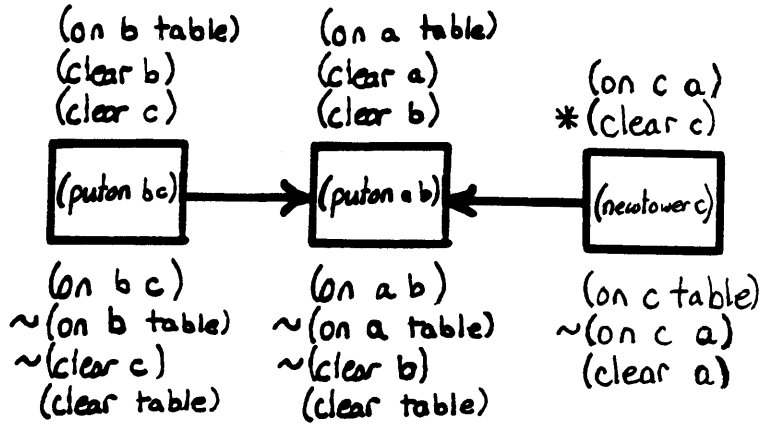
Unachieved preconditions are starred. The on preconditions are achieved by being constrained to codesignate with propositions in the initial situation:



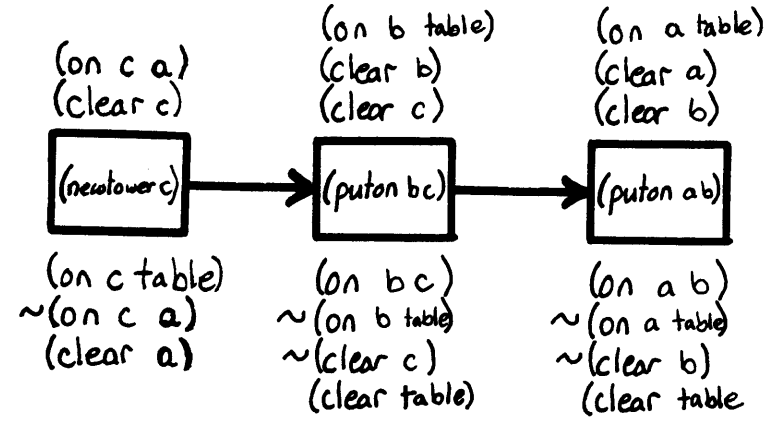
The precondition (clear b) of the second step is unachieved because the first step is possibly before the second and denies the precondition. TWEAK achieves (clear b) by promotion of (puton a b):



Then the precondition (clear a) of (puton a b) is achieved by addition of a newtower step. This has a precondition (on x a) which is satisfied by being constrained to codesignate with (on c a), which is true in the initial situation.



Finally TWEAK achieves (clear c) by promotion of (puton b c):



Chapter 3

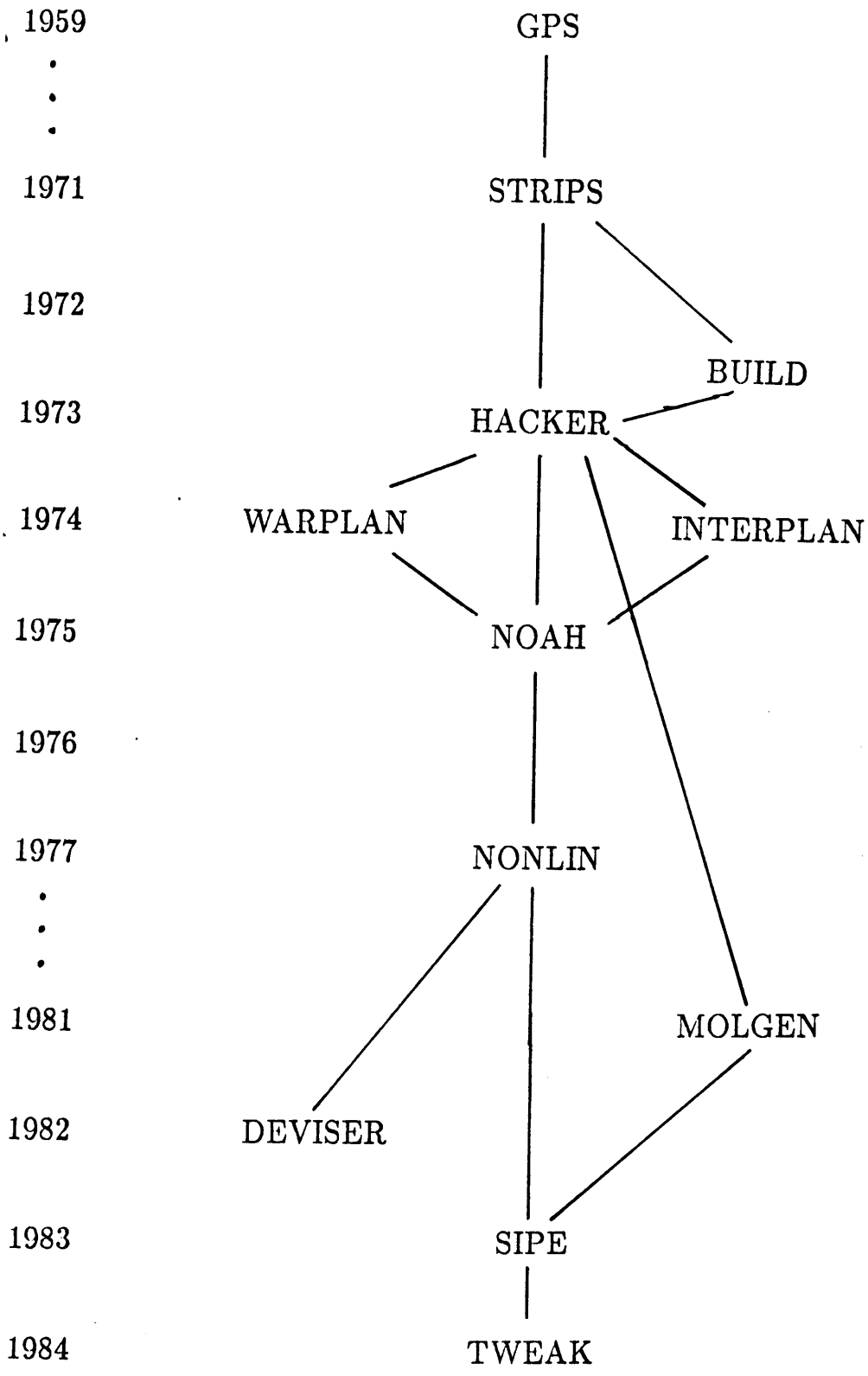
Past and future planning research

The three main points of this chapter are that in retrospect all domain-independent conjunctive planners work the same way; that the action representation which they depend on is inadequate for real-world planning; and that desirable extensions to this action representation make planning exponentially harder. It is much longer than such sections are in typical AI papers because domain-independent conjunctive planning is unusual as a subfield of AI in showing a clear line of researchers duplicating and building on each other's work. Science is supposed to be like that, but for the most part AI hasn't been.

I restrict attention to domain-independent conjunctive planning, ignoring planners and parts thereof that are domain-dependent or non-conjunctive. This may seem unfair at times. There are two previous survey articles on this topic, [46] and [61]. The facts I consider are much the same as those covered by the other papers; my analyses of many points are different.

The first section in this chapter is a historical overview of domain-independent conjunctive planning, showing how different planners build on one another, with particular emphasis on the history of the ideas embodied in TWEAK. The other three sections are devoted to the history and future of the three levels of a conjunctive planner: representation, plan modification operations, and top-level search strategies. The most interesting suggestions for future research are in section 3.2.1 on action representation; the most interesting analysis of past work is in section 3.3 on plan modification operations.

Planning: A Family Tree



3.1 Chronology

Two important "prehistorical" non-conjunctive planners introduced techniques that underlie all the conjunctive planning work. GPS [35], due to Allen Newell, J. C. Shaw, and Herbert Simon, introduced means-ends analysis, which is to say step addition or subgoaling: solving problems by applying an operator that would achieve some goal of the problem, and taking the preconditions of the operator as new goals. STRIPS [14], due to Richard Fikes and Nils Nilsson, contributed the action model—in which steps have postconditions that are the only things that get changed by the step—that is used by all domain-independent conjunctive planners.

Domain-independent conjunctive planning begins in 1973 with Gerald Jay Sussman's HACKER [55]. Sussman ended his thesis with the problem described in section 1.1, due to Allen Brown but widely known as "the Sussman anomaly," which HACKER could not solve without resort to what Sussman called a "hack".

The urge to find a clean solution to the Sussman anomaly drove a series of rapid developments over the next four years. David Warren's WARPLAN [67] and Austin Tate's INTERPLAN [56,57,58], both of 1974, cleaned up Sussman's ad-hoc "hack": promotion, in fact. Richard Waldinger [66] further generalized promotion.

In 1975 came Earl Sacerdoti's NOAH [44,45], the first nonlinear planner. Besides his improvement in the representation of plans, Sacerdoti substantially expanded the set of plan modification operations. Tate (the same author of INTERPLAN) improved on NOAH in 1976. NONLIN [59,60] had a backtracking top-level control structure, so that it could find plans after NOAH would get stuck, and added to NOAH's set of plan modification operations.

After 1976, there was a great drought for many years. During this period, there was one important piece of work on non-conjunctive planning: Mark Stefik's MOLGEN [49] made constraints a central technique in planning for the first time. Conjunctive planning was not advanced until a new spurt of work beginning in 1982.

All the new conjunctive planners were NOAH-based. Several researchers [2,31,32,65] extended NOAH by improving the representation of time, in quite different ways. (These improvements have not been incorporated in TWEAK.) David Wilkin's SIPE [71,72] used MOLGEN-like constraints and incorporated a new technique for detecting clobbering.

3.2 Representation

A planner must represent events in time, actions the agent can take, and the world and the objects in it. Domain-independent planners all base their representations on those of STRIPS, and with the exception of the introduction of constraints, have not progressed much beyond that framework. Therefore, this section is more concerned with future than with past work.

The rest of this section is divided into four decreasingly interesting subsections.

The first discusses action representation; the second, time; the third, codesignation constraints; and the fourth, miscellany.

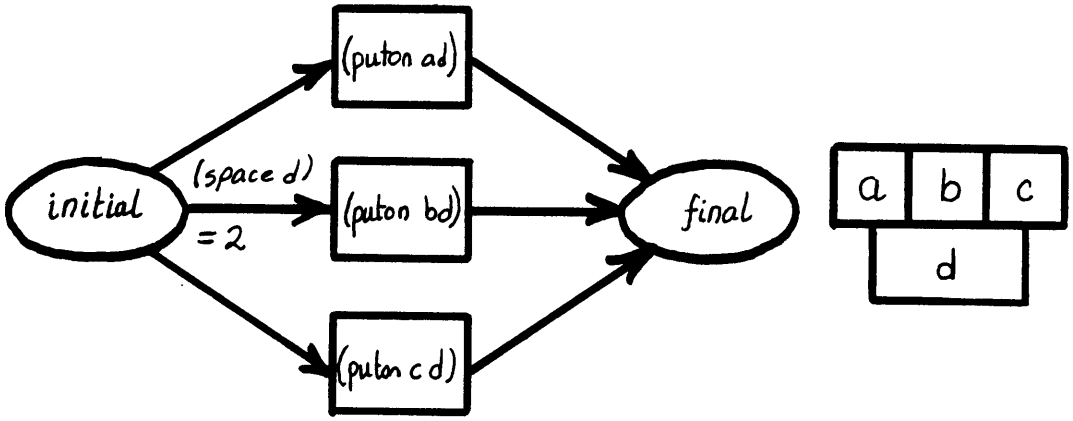
3.2.1 Actions

TWEAK has an impoverished representation for actions; future work must correct this. The action representation does not allow for indirect effects or for the effects of an action to depend on the situation in which it is applied. Without these restrictions, the modal truth criterion would fail, and TWEAK would be no longer complete and perhaps not correct. These problems are largely a reflection of the state of the art and are not specific to TWEAK.

The restrictions on action representation make TWEAK almost useless as a real-world planner. It is barely possible to formalize the cubical blocks world in the TWEAK representation; HACKER's blocks world, with different sized blocks, can not be represented. I see planning with action representations that can express real-world problems as the foremost challenge for future domain-independent planning research. In the remainder of this section, I will explain the two major restrictions, the reason the truth criterion fails in each case, and some approaches to extending TWEAK for a richer action representation.

The action representation does not allow the effects of actions to depend on the situation in which they are applied. Consider a blocks world in which zero, one, or two blocks can be on any given block. (This example and its analysis are due to David McAllester, personal communication.) Every block still must be on zero or one other block. We need a function space that takes a block as an argument and returns an integer between zero and two inclusive that tells how much room is left on top of the block. A precondition of (puton a b) is that (space b) be greater than zero, and the corresponding postcondition is that (space b) be one less than before. TWEAK can not represent this puton; a representation with conditional postconditions or postconditions that are functionally dependent on the input situation could.

If TWEAK were extended to express this action, the modal truth criterion would fail. An example is provided by a plan with three unordered steps, (puton a d), (puton b d), and (puton c d), and with (space d) being two in the initial situation.



A precondition of each step is that (space d) be at least one. Let us ask whether this precondition p is achieved in the input situation i of (puton a d). According to the modal truth criterion, it is achieved so long as there is a situation t (the initial situation will do) necessarily before i in which p is necessarily asserted (all true) and that there is not even possibly a step between t and i that denies p. Candidate clobbering steps are (puton b d) and (puton c d); they are possibly between t and i. Does one deny p? No, because in i there is space for two, and each step only decrements the space by one. Yet the two steps act synergistically to clobber p. This possibility is not accounted for in the modal truth criterion.

Another restriction is that all changes made by an action must be explicitly represented as postconditions; many actions can not be formalized in this representation. For example, if block b is on block a and we move a from room_1 to room_2, b will also move. This effect could be captured in an action representation in which deduction was allowed within situations, so that propositions logically following from postconditions would be considered true in the output situation of a step. Call the set of all propositions that follow from another set the deductive closure of the second set. The semantics of executing a step in a situation is to negate all the propositions in the deductive closure of the postconditions and remove that set from the input situation, add the postconditions to this result, and take the deductive closure of all that. It is again possible for two steps to act synergistically to assert or deny a proposition: if $q \wedge r \Rightarrow p$ and one step asserts q and the other r, together they assert p (equivalently deny $\sim p$). This is the reason that TWEAK requires all propositions to be atomic. Non-atomic propositions could be used, but would be simply treated as literals; the logical operators can't get their usual semantics without deduction.

There are two obvious ways of modifying the modal truth criterion to handle these problems of synergy. One is to consider all the completions of a plan. In a linear plan, the state of the world is completely known at all times, and no synergy is possible. Unfortunately, there are exponentially many linearizations of an incomplete plan, and

so this approach, which amounts to reverting to linear planning, is not efficient. Another alternative is to consider sets of steps in trying to find establishers and clobberers. But again there are exponentially many subsets of a set of steps.

Among nonlinear planners, only SIPE allows derived effects or dependency of effects on the input situation of a step. SIPE's treatment of these features is incomplete and not generally correct. This is reasonable, as SIPE represents the state of the art in engineering, rather than a formal theory of planning.

Waldinger's planner allows for derived effects and for effects that depend on the input situation of a step. Because his planner is linear, the problems of synergy do not arise. To determine whether p holds in the output situation of a step S, he asks whether it or its negation is asserted there. If not, p is *regressed* over S. A regression, q, is computed from p and S such that q being true in the input situation of S guarantees that p is true in the output situation. Now the rule can be applied recursively to q and the output situation of the step preceding S.

There are two difficulties with Waldinger's approach. First, it depends on linearity; in a nonlinear plan there isn't a unique step preceding S to recurse to. (Pednault [36] is able to extend Waldinger's technique to a certain restricted form of nonlinearity.) In section 3.2.2 I will suggest that linear planning is exponentially less efficient than nonlinear planning.

The second problem is that it is not obvious how to compute the regression of p given a step to pass it back over. Waldinger does not address this problem; his planner apparently was given specific techniques which worked only for particular action types. Rosenschein's planner [41] has a general procedure for computing regressions, which was further extended by Kautz [24,25]. Because these systems incorporated complete deduction engines, they proved to be unworkably inefficient.

The regression formulation makes clearer how TWEAK depends on the simplicity of its action representation: the essential factor is that propositions are unchanged by regression. Extensions to the action representation that preserve this property would be safe. I haven't been able to find any such extensions, however.

The essential difficulty with extended action representation is the *frame problem* [23,30,37]. The frame problem is traditionally stated as that of discovering what propositions are changed by an action; this is useful in practice in order to discover whether a proposition holds in a situation. Thus the frame problem can be viewed as that of finding an efficiently-implementable truth criterion. The following theorem suggests that this may be impossible in general:

Intractability theorem: The problem of determining whether a proposition is necessarily true in a nonlinear plan whose action representation is sufficiently strong to represent conditional actions, dependency of effects on input situations, or derived side effects is NP-hard.

A somewhat related theorem is the following:

Second undecidability theorem: Planning is undecidable even with a finite initial situation if the action representation is extended to represent actions whose effects are a function of their input situation.

What are we to make of these theorems? Naively, they suggest that writing planners for extended action representations is a quixotic enterprise. In the conclusion to this report, I will make a radical suggestion in this regard. There are loosely three ways out, however. We might hope for the best, relax the correctness requirement on a planner, or relax the generality requirement. Hoping for the best amounts to arguing that for the particular cases that come up in practice, extensions to current planning techniques will happen to be efficient. My intuition is that this is not the case, but the issue is an empirical one. Relaxing the correctness requirement would produce heuristic planners that sometimes produce plans that don't quite work. A number of current planning systems fit this category. In the remainder of this section, I'll discuss the third possibility, relaxing the generality requirement.

I have examined a number of specific domains, and found that for each of them it was easy to find an efficient truth criterion, but that these criteria were quite different. Perhaps then we should give up on domain-independent planning: the user of a planner must specify, together with the set of available actions, truth criteria to be used.

At the expense of some scruffiness, we can do better. In other work [5], I have been developing a theory of *intermediate techniques*, which are neither completely general, nor completely domain-specific. *Cognitive cliches*, formal structures occurring in many domains, have attached to them intermediate competence that is specific not to a domain, but to a cliche. Intermediate competence is applied by identifying instances of the associated cliche in the world. A cliche-based system has to know something about the domain it is running in, but is still domain-independent, since any cliche may show up in any domain. I envision a cliche-based constraint-posting planner for extended action representations which would have truth criteria specific to cliches that operators in the world might instantiate. A planner with truth criteria for a few dozen cliches might well cover most interesting domains.

For an example of planning with cliches, consider resources. An instance of the *resource cliche* consists of a state variable in the world which holds a quantity in some total order, together with at least one *consumer operator*, which decreases, relative to the order, the value of the state variable, and at least one *dependent operator*, which has as a precondition that the state variable have a value greater than some threshold. There may also be *producer operators* that increase the value of the state variable. (I'm using the term "resource" differently than Wilkins does.) Resources are found in many domains; puton in HACKER's blocks world is both a consumer and a dependent of the space on any given block.

Associated with the resource cliche is a truth criterion. The value of the state variable in a situation s is no less than its value in situation t necessarily before s, minus the sum of the amounts of decrements due to consumers possibly between t and s, plus the sum of the amount of increments due to producers necessarily between t and

s. In fact, it is often possible to prove a higher lower bound than this; [38] describes a clever polynomial algorithm using network flow techniques that computes the exact least value the state variable could take on.

From this truth criterion we can derive three plan modification operations: the precondition of a dependent operator can be achieved, if it is not already, by adding producers between t and s, by constraining consumers possibly between t and s to be before t or after s, or by constraining the amounts of consumption or production to be respectively small or large. In the HACKER blocks world, adding producers of space amounts to the "punting" strategy (see page 30), and increasing the amount they increase space by suggests the "compacting" strategy.

Unfortunately, it turns out that planning with resources is NP-complete; it appears as "sequencing to minimize maximum cumulative cost" in [17]. However, the polynomial criterion and the achievement procedure will be extremely useful if additional domain constraints can be exploited to limit search.

3.2.2 Time

The representation of time is crucial to planning: a plan is really a representation of part of the future. The biggest advance in domain-independent conjunctive planning was probably the recognition that the time order can be partial, at least until execution. This observation first appears in print in [67, p. 16], but the first implementation was in NOAH.

I believe that nonlinear planning is potentially exponentially more efficient than linear planning. In an extended action representation, linear planning avoids the potentially exponential amount of work required to compute the truth criterion; but the same exponential shows up in a different guise as the potentially exponentially greater amount of backtracking required to find the correct plan. That more search is required for linear planners is supported by intuition and empirical evidence, but has never been formally proved. It appears to be tricky, because it requires diagonalizing over the search strategy of the linear planner. Thinking about metalevel planning in attempting the proof might be good for this reason.

I have simplified the representation of plans from those used in NOAH and NONLIN. Those planners represent plans as directed acyclic graphs in which there are many different types of nodes, only a few of which represent anything much. My plans are simply partial orders on steps.

Much of the post-drought planning research in the last few years has focused on overlapping actions. All the old planners assume that actions are instantaneous and atomic; in the real world most actions take time, and several can happen at once. Steven Vere's DEVISER [65] treats actions as temporal intervals with numerical endpoints. James Allen and Johannes Koomen's planner [2] also treats actions as intervals, but is based on Allen's non-numerical time logic. Drew McDermott [31] suggests using a time logic based on branching futures as a basis for planning. These approaches are combined in Miller et al.'s FORBIN [32]. In all these formulations, several actions can be

executed in parallel. It would be interesting to analyze these planners in the same way I have analyzed TWEAK: particularly, to find a provable truth criterion that accounts for overlapping actions and to see what plan modification operations it engenders.

Allen's time logic can represent the constraint that two actions be disjoint in time without committing to which order they are to be performed in. This makes it possible to defer the choice of declobbering operation further than can be done in TWEAK. Promotion and demotion can be combined into a single constraint, which does not commit to which is to be used. This decrease in commitment may result in less search. Most generally, one could represent time propositionally, allowing general disjunctions between several possible constraints to be expressed. This would trade off commitment against the cost of deducing facts about a particular incomplete plan.

Plans are like programs in many ways; but programs have conditionals, iterations, and dataflows, which domain-independent planners have not for the most part been able to generate. A version of WARPLAN generated conditional plans [68], as did Rosenschein's planner [41]; see page 31. Van Baalen [64] describes a planner that uses numerical cost information to generate conditional plans. NOAH had a feature for representing simple iterations; however, this representation does not allow declobbering between steps inside the loop and steps outside, and so can not be called conjunctive. The Programmer's Apprentice [39,40] uses a "plan calculus" historically derived in part from NOAH, which can represent conditionals, loops, and dataflow. Lansky [27] describes GEM, a plan representation derived from studies of concurrent programming systems, which can express a constraint such as "customers are served in the order in which they make requests" which is difficult in other representations. No program synthesizer has yet been written using the Programmer's Apprentice or GEM representations.

3.2.3 Codesignation constraints

MOLGEN was the first planner to highlight the use of constraints; its author, Stefik, introduced the term "constraint posting." Constraints in MOLGEN are arbitrary predicates possibly on several variables. MOLGEN performs three operations on constraints: formulation, propagation, and establishment. Formulation is making new constraints, propagation creates new from old constraints, and establishment is binding variables to values. MOLGEN was the first planner to do propagation; unfortunately his propagation techniques are domain-dependent and not even described in his thesis. Stefik describes a "build or buy decision" in achieving a goal involving a variable: either one can bind it to a constant already appearing in the plan ("buy"), or to a new constant ("build"). In this case it is often necessary to introduce new steps whose postconditions involve the constant, so as to guarantee properties of it. MOLGEN was first to introduce new steps to satisfy a constraint.

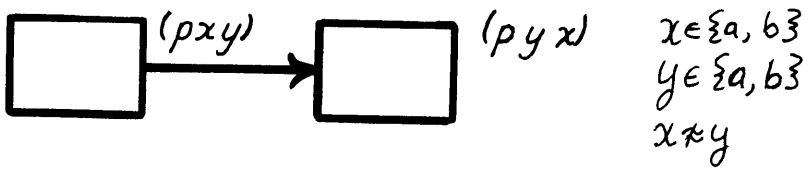
It is little recognized that HACKER used codesignation constraints. They were implemented as special type preconditions on variables. HACKER's clever techniques for achieving goals with variables make use of the CONNIVER [54] context mechanism and have not been duplicated since. However, only the "buy" option was considered,

and it is not clear how general the implementation was.

SIPE's constraints are modeled on MOLGEN's. SIPE's truth criterion takes into account possible truths resulting from possibly codesignating propositions. Like TWEAK, SIPE propagates constraints only via a codesignation relation.

TWEAK uses only codesignation constraints, because preconditions already can represent predicates on variables, so that there is little loss in expressive power. If one looks at the way constraints are used in planning, almost all constraints correspond very naturally to preconditions of steps. There are some exceptions to this in MOLGEN, all of them constraints that have been created via propagation. Stefik's build or buy decision translates in the TWEAK framework into step addition versus simple achievement. Since preconditions are associated with times, predicates on variables are also; this solves problems MOLGEN had with time representation.

The difference in expressive power between MOLGEN or SIPE and TWEAK is that TWEAK can not restrict the range of a variable to a finite set. There are two reasons I haven't put range restrictions into TWEAK: because constraint computations then become NP-complete (proof by reduction from graph coloring); and more seriously because the truth criterion fails if I allow them. This "pathological" plan illustrates the problem:



Here the codesignation constraints require that either x codesignate with a and y with b, or vice versa; either way $p = (\mathsf{p\ a\ b})$ holds in the final situation. Yet p is not necessarily asserted by any particular step.

Variables are needed in the blocks world in which they originated for a "deep" reason. In [4] I describe a problem solver that uses a TWEAK-like planner as a subroutine. This problem solver views puton as both a POP and a PUSH. It is the PUSH aspect that is exploited in achieving on goals, and the POP aspect that is exploited in achieving clear goals. When puton is viewed as a POP, there is no explanation of what the second argument (the place to put the block moved) is for. So the problem solver uses a variable to leave the second argument unspecified. Whatever value the second argument takes on, the puton acts as a POP.

3.2.4 Other problems for future research

A 1972 paper by Fikes, Hart, and Nilsson [15] lists a number of open research areas in planning. Most are still virtually untouched.

Representations of the world using non-assertional datastructures are simpler, more efficient, and better reflect its structure than the assertional databases used in current domain-independent planners. Many domain-specific planners use such representations effectively, and I see no inherent difficulty in using such simulation structures in linear domain-independent planning. In constraint-posting planning the state of the world is not completely defined at all times; this is easy to implement using assertional databases, in which it is easy to represent unknown truth values. It is much harder to represent partial knowledge with non-assertional datastructures. One approach to this problem is outlined in [43]. This or perhaps a hybrid technique, using both assertions and more direct representations, may lead to simpler, more powerful and efficient planners.

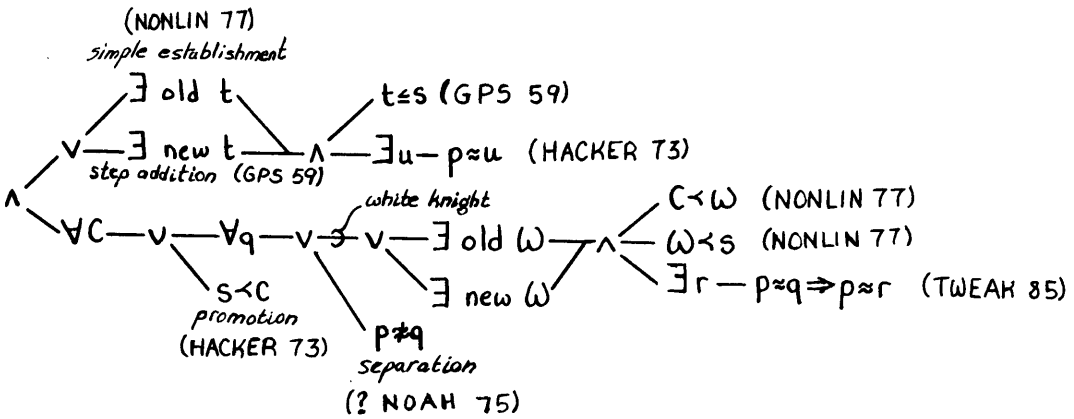
An important open problem in planning is coping with events that are not planned for. These might be the actions of other agents or spontaneous law-governed physical happenings. DEVISER and Allen's planner can plan around "scheduled" unplanned events: these are specific events that will occur at known times. Many problem solvers have plan executives that, when unexpected events occur, call the planner to derive a new plan from the altered state of the world; [73], for example. In the real world, which has tigers in it, that isn't good enough; you have to prepare for contingencies during planning. Such plans need conditionals.

Related problems are planning with incomplete or incorrect knowledge of the world and with unreliable primitive actions. Another area of current research is in planning for multiple agents or multiple effectors that have to be synchronized [15,16,42].

3.3 Making a plan achieve a goal

In this section, I treat first linear planners and then nonlinear planners. That isn't quite the chronological order, as some linear planners postdate NOAH. There are two interesting points to the section: one is the way the individual plan modification operations were developed by generalization, splitting, and merging. The other is to see that all conjunctive domain-independent planners work in substantially the same way, though they look very different, using apparently unrelated datastructures and algorithms. As time went by, features were added and alternative implementations were tried, but the fundamentals are unchanged from HACKER down to TWEAK. This has not been generally realized, even by the people who wrote the planners. Forcing all the algorithms into the vocabulary of TWEAK modification operations makes them easy to compare. However, many of the early planning papers are very difficult to read, and some of what follows may be inaccurate in detail.

This diagram summarizes the section, illustrating which parts of the achievement procedure were invented when:



The italic labels name plan modification operations. *Demotion* is a particular subcase of declobbering by white knight that is symmetric with promotion. Demotion is the case in which t is chosen to be the output situation of W, so that the clobberer is nullified by moving it before the establisher.

The most important thing to understand about linear planners is that they work just the same way as nonlinear ones, except that the representation is awkward. The basic operation of all the linear planners is analogous to promotion. In a nonlinear planner this just adds temporal constraints; in a linear planner, a step must be picked up and moved to a different position in the plan.

There are two different versions of promotion that appear in linear planners. The first, ***individual promotion***, moves the clobberer forward over the clobberee. (Alternatively, the clobberee could be moved backward before the clobberer; this is an uninterestingly different operation.) In a nonlinear planner, promotion automatically also puts everything before the clobberee before the clobberer and vice versa; individual promotion doesn't generally do so, with the result that a step can be separated from the steps that were to achieve its preconditions, so that they must be reachieved. ***Block promotion*** moves the clobberer, together with the steps that achieve its preconditions, and the steps that achieve their preconditions, transitively, as a block. This implies a ***strong linearity assumption***: not only that the plan can be totally ordered, but also that if you have goals g and h and S achieves g and T achieves h and S is before T, then all the steps that achieve preconditions of S are before all the steps that achieve preconditions of T. In other words, the time order must respect the subgoaling hierarchy. Using only block promotion, it is impossible to solve optimally the Sussman anomaly problem. The optimal solution involves three steps: (puton c table), (puton b c), (puton a b). This

plan violates the strong linearity assumption: the first step achieves the precondition (clear a) for the last step, but the middle step is not achieving a precondition of the last, but rather one of the top-level goals.

HACKER has, in effect, four plan modification operations. Step addition is used initially on each of the conjunct goals, and the resulting steps are arbitrarily linearly ordered. HACKER recognizes four bug types, each of which has a corresponding plan modification operation. "Prerequisite Missing" is a precondition that is not true anywhere before it is needed, and is patched with step addition. "Prerequisite Clobbers Brother Goal" is just clobbering, and block promotion is applied. "Prerequisite Conflict Brothers" is a "doublecross": a pair of steps each of which clobbers the other. HACKER has a plan modification operation for this which does not appear in any other planner: a RESOLVE expert is called, which replaces the two steps with a single step that achieves all the goals the two steps together were intended to achieve. In practice, it seems that the only cases the RESOLVE expert could handle were pairs of steps that achieved the goals (spacefor a c) and (spacefor b c) (Sussman's blocks world allows more than one block on a given block). The expert would replace the two steps with a subplan that achieved (spacefor (both a b) c).

HACKER had many other nifty planning techniques that somehow got lost in the sands of time. For example, HACKER's addition operation is different from those of all subsequent planners. Addition in later planners uses one of the possibly several steps that could achieve a goal, perhaps saving the others as backtrack alternatives. HACKER doesn't backtrack, but it does add the alternate steps in later when one doesn't work. This leads to the fourth bug type, "Strategy Conflict Brothers," in which a step in one strategy (alternative achieving step) clobbers a precondition of another, later strategy. In this case, HACKER applies promotion. This "multiple addition" operation has many interesting properties. The principal use of it is in achieving (spacefor a b); the two strategies are "compacting" the blocks on top of b and "punting" blocks off of b that don't need to be there. Although either of these strategies may achieve an unachieved spacefor goal, neither is guaranteed to. Yet, if executed in the right order (punt then compact), they make space if it is possible to do so. This sort of synergy and partial goal fulfillment has never been duplicated.

Sussman called Allen Brown's problem "anomalous" because it could not be solved using block promotion. He presents a solution using individual promotion, but regards this as a "hack." Why? Sussman viewed HACKER as an automatic programming system, constructing programs, not plans. A conjunction (of the original goals or of the preconditions to a step) is achieved via a single subroutine. Promotion in HACKER was confined to permuting the order of lines of a subroutine; this amounts to block promotion, since subroutines encapsulate the subgoal hierarchy. In order to solve the Sussman anomaly, one must move program steps across subroutine boundaries, which HACKER wouldn't do.

It's a pity, though, that the view of planning as automatic programming got lost in the shuffle. HACKER's performance in any given domain would improve as time went

by, because the programs it wrote could be re-used on new problems. Sussman describes techniques for generalization and subroutinization of programs so that less planning would need to be done later. Compilation can be viewed as constant-folding the source code into the interpreter; HACKER in effect constant-folded classes of problems into the planner. It would be interesting to build a problem solver that constant-folded into TWEAK, and incorporated what has been learned in the past ten years about generalization.

WARPLAN has two plan modification operations, step addition and an operation that combines addition with individual promotion. The latter operation ("regression") is to find a step that achieves the goal, then to search backward from the end for a place in the plan where the step can be put without being clobbered. WARPLAN was able to solve the anomalous problem because it doesn't make the strong linearity assumption; it represents plans as flat orders, without hierarchy.

Waldinger [66] generalizes Warren's technique by allowing the regression of a goal to be computed from the step it is being moved back over (as described in section 3.2.1).

INTERPLAN has three plan modification operations: step addition and both versions of promotion (not combined with addition).

Rosenschein [41] describes a linear planner that uses both promotion and demotion. He also has an operation for introducing conditionals (if-then-else branches) into plans: it chooses an arbitrary proposition not provable or disprovable from the initial situation, and puts a branch on this proposition at the beginning of the plan. In unpublished work he has improved this technique so that the condition to branch on can be found deterministically.

With NOAH comes the great explosion in the set of plan modification operations. NOAH classifies clobberings into three sets: in the first, two steps each clobber the other (a "double cross"); in the second, the clobberer and clobberee are unordered; and in the third, the clobberer is before the clobberee. The case (an "n-cross") in which a set of more than two steps clobber each other, arranged in a cycle, is neglected. I don't understand Sacerdoti's explanation of the plan modification operation to patch double crosses; it seems to be a version of step addition, possibly combined with separation. The other two cases are handled by promotion and demotion with the addition of a white knight step, respectively. Sacerdoti's description of his promotion operation is inconsistent; he gives a description of a correct operation, but also describes one based on a datastructure called a Table of Multiple Effects which is incorrect (as noticed by Tate).

"Eliminate Redundant Preconditions" is a step removal operation. Step removal is useful because NOAH does not have a simple establishment operation. Thus, if two steps have the same precondition, they may both be achieved by addition, and then one of the two steps removed. "Use Existing Objects" binds variables to constants. Sacerdoti is very unclear on when it is applied and how the binding is chosen. NOAH has a simple but entirely adequate technique for achieving a disjunctive goal. Each of the disjuncts is planned for, until it is clear that one can be achieved; then an operation

is applied that removes the plan fragments for the other disjuncts.

Sacerdoti presents two "task specific" plan modification operations. "Tool Gathering" optimizes plans relative to a notion of the cost of performing correct plans: a correct plan may be made into a better, still correct one by some re-orderings. "Limitations of an Apprentice" compensates for the inexpressibility in his action representation of many kinds of actions. The example he gives is very similar (a resource conflict, requiring a global view to do declobbering) to the blocks world example I analyze in section 3.2.1. Unfortunately, the details of the operation are not given.

NONLIN was the first planner to use simple establishment. NONLIN also uses addition, promotion, and demotion.

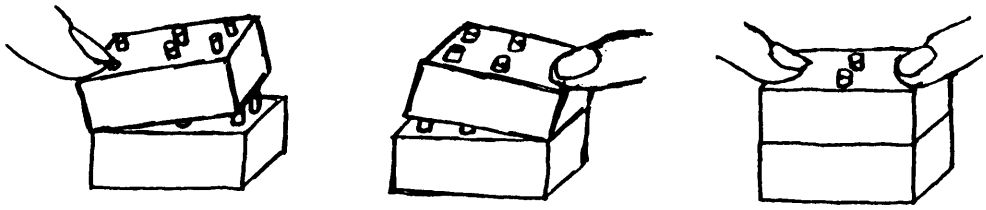
SIPE introduced no new modification operations, but it does have a new technique for detecting clobbering. A particularly common sort of precondition is what Wilkins terms a *resource*: a binary variable that must be set to one value ("available") for an operation to be applicable, and which is set to a different value during the operation, then "released" or reset at the end. Two unordered steps that try to use the same resource clobber each other; SIPE then applies promotion. The techniques SIPE uses for resource clobbering detection are only heuristic; more work is needed to understand this maneuver.

Separation as a plan modification operation may appear first in TWEAK. I suspect that NOAH's double-cross removal operation, which I don't understand, may combine step addition with separation; apart from this, there seems to be no precedent.

TWEAK is the first planner to use declobbering by white knight in the general case in which the white knight is distinct from the establishing situation t. This maneuver is actually pretty useless, because either the white knight will turn out to assert the goal, in which case we'd have done as well to use its output situation as an establisher, or else it won't, in which case we might as well have used separation to defeat the clobberer. The only possible advantage is slightly less commitment, and so possibly a little less backtracking. This plan modification, like all the others, falls out of interpreting the modal truth criterion as a nondeterministic procedure, and corresponds to the necessarily but odd clause about white knights. My implementation never actually uses it.

Because the modal truth criterion is sufficient as well as necessary, there are no more plan modification operations possible without extending the range of represented actions. Once that is done, new operations will be possible; again they can be derived from the truth criteria for the new representations. For example, the pathological plan on page 27 suggests an operation of "establishment by separation": if the plan did not have the constraint $x \not\approx y$, adding this constraint would achieve p in the final situation.

If the representation of time is extended to allow overlapping actions, a new operation ("simultaneous establishment") is possible. In this figure,



someone is trying to stick two LEGO blocks together. Pushing the top block down on either end will tend to make the block pivot around the center pair of posts, so that the two blocks do not mate. The problem can be solved by performing both actions simultaneously, pushing with a finger at each end. This is similar to HACKER's compacting/punting synergy except that it depends on the two strategies being applied simultaneously. (Another solution is to push in the middle, but that's not very interesting.)

Kristian Hammond [18] describes WOK, a planner which although domain-specific and non-conjunctive has interesting things to say about goal interactions (the class of effects that includes clobbering). WOK achieves goals by introducing interactions of known sorts. This is the antithesis of the linear strategy: rather than assuming as a first approximation that goals don't interact, synergistic interactions are used as a basic tool for achieving goals. It is unclear how this approach can be applied to conjunctive planning, but it may be a useful line of future research. [70] also describes constructive use of goal interactions.

3.4 The top-level control structure

The top-level control structure of almost every domain-independent conjunctive planner is search. Search control is the aspect of domain-independent conjunctive planning that is understood least. Most of the domains to which domain-independent conjunctive planning has been applied have been forgiving: if more than one plan modification operation is applicable to a clobbering or unestablished goal, any of the possibilities will probably do. Thus, it hasn't been necessary to devote a lot of thought to which to choose. However, in real domains the choices probably are critical, and a lot of schemes have been proposed for making them. Since none of these has been adequately tested, little is known about which is best.

Almost every planner has a distinct control structure. I've loosely grouped them in seven classes, ordered roughly by the complexity of the backtracking algorithm. The classes are no backtracking, explicitly represented alternatives, dependency-directed modification, chronological backtracking, dependency-directed backtracking, heuristic search, and metaplanning. Many of the planners I discuss actually fit into several of these classes.

The simplest control structure avoids backtracking altogether. Plan modification operations are applied in a fixed order according to fixed criteria until a correct plan is found or it is no longer possible to apply operations. This is not as bad as it sounds, because you can usually make a good guess as to which modification operation to apply: usually, one should prefer simple achievement to step addition, for example. HACKER uses this approach. NOAH comes very close; it backs up only from alternative choices of variable bindings. That NOAH solved many difficult problems shows that the choice of control structure is unimportant in some domains.

A very simple solution to the problem of which modification operation to apply is to choose all applicable ones, splitting the plan into several explicitly represented copies. No planner fits altogether in this class as it is very inefficient in general. If some additional principle decides whether for a given choice to use this technique or to use search, the splitting technique may be useful. SIPE and a planning framework described by Hayes-Roth et al. [20] take this approach.

The simplest backtracking scheme is chronological: when a choice has to be made, one is chosen by some means and the others are saved away. If the plan can not be extended to a solution by further modification, failure is signalled. The most recent choice point is backed up to, and an alternative for the choice is used. When no choices remain, the next most recent choice point is backed up to, and so on. WARPLAN, INTERPLAN, and SIPE use chronological backtracking.

Chronological backtracking can result in the exploration of more blind alleys than necessary. Dependency directed backtracking backs up at failure not to the most recent choice point, but to one responsible for the failure. For a discussion of dependency-directed backtracking in general, see [11] and [48]. The first planner to use dependency-direction was Hayes's 1975 route planner [19], which was not conjunctive. Hayes used backtracking to recover only from execution error, rather than from planning error (dead ends) as does TWEAK, although he explicitly considered the latter possibility. Thus his implemented control structure can be termed "dependency-directed modification" rather than backtracking. Hayes's conception of dependency-directed backtracking predates and seems to be independent of its discovery by Stallman and Sussman [48], to whom it is usually credited.

Daniel [8] added dependency-directed backtracking to NONLIN. The same year deKleer et al. [9] described a dependency-directed linear planner. London's planner [28] represents plans and world states using a TMS, the utility underlying dependency-directed backtracking, but apparently does not use the TMS for backtracking. To do so, the decisions taken in choosing one rather than another plan modification operation—the metaplan—must be represented, and London did not do so.

Heuristic search uses some numerical estimate of "goodness" to decide which order to try choices in. INTERPLAN and NONLIN use heuristics to control their chronological search. Since making a wrong choice can result in searching a large dead-end subtree, it would be nice to eliminate wrong choices without having to explore their consequences. Kibler and Morris [26] present a control scheme based on negative search heuristics that

prune obviously bad choices. However, these heuristics are domain-specific for the blocks world. Siklossy and Roach [47] use a similar strategy. Corkill [6] describes a NOAH-like planner in which control is distributed among several message-passing processors.

All the control structures discussed so far (with the possible exception of heuristic search) are "syntactic": they don't depend on the specifics of the plan being constructed, but blindly apply some simple algorithm for choosing among alternatives without considering what those alternatives are. Since control of planning is very hard, such methods may be inherently weak; perhaps we should apply the full power of a problem solver to choosing what to do next. This is the *metaplanning* approach. There is an increasing literature on this [3,7,12,50,69,70] most of which is very vague. I'll discuss just two metaplanning systems. Doyle's unimplemented SEAN uses (another copy of) the same planner to do metaplanning as to do planning about the domain. The metaplanner in turn is controlled by an identical metametaplanner and so on; Doyle discusses ways to implement this apparently infinite regress.

MOLGEN has only one level of metaplanning, and the metaplanner is quite unlike the domain-level planner. The domain-level planner creates plans for MOLGEN's domain, genetics experiment planning. It has operations that are analogous to the plan modification operations of TWEAK. These operations are selected by a metaplanner which chooses among plan modification operations. The metaplanner is very simple; it's perhaps grandiose to call it a planner at all.

The idea of using a copy of TWEAK as a metaplanner is attractive: the plan modification operations can be thought of as having well-defined preconditions (that the constraints they impose not conflict with the existing ones, or that a suitable step exists to achieve a goal in the case of addition) and postconditions (the insertion of the new constraint or step). Unfortunately, TWEAK's action representation is too weak to represent the plan modification operations.

Planners can be classified along a dimension orthogonal to search strategy, that of technique used for recover from execution failures. This isn't part of planning proper, but many systems interleave planning with (simulated or actual) execution so that effectively a non-backtracking planner performs search, failing during execution rather than planning, and then returning to the planner to obtain a new plan to recover. HACKER uses this approach. NOAH proper doesn't, but its planner is connected to an execution system that re-invokes the planner after execution failure, so that the system as a whole can be put in this class. HACKER makes use of CONNIVER techniques similar to dependency-direction in order to figure out which planning decision was responsible for the failure and to try another alternative in the choice.

Planning and AI language design have strongly influenced each other. Many of the planners that do search inherit their search discipline from the language they were written in, and many AI languages were designed to make writing the top-level control of planners easier. PLANNER [21,22] was intended as a language for writing planners in; it was the first to supply backtracking automatically. HACKER and BUILD, a clever domain-specific blocks-world planner [13], were written in and depend heavily on the

abilities of CONNIVER [54]. CONNIVER was written in reaction to the difficulties with chronological backtracking in PLANNER. WARPLAN inherits its search from PROLOG. The planner of deKleer et al. was the first program written in AMORD, and inherits its dependency-directed backtracking from AMORD's TMS [10]. TWEAK, too, inherits its dependency-directed backtracking from Dependency-Directed Lisp, a language specifically designed for TWEAK. DDL looks like ordinary Lisp but has an implicit dependency-directed backtracking control structure. It will be described in a forthcoming paper.

Chapter 4

Conclusions

Perhaps the most important contribution of this report is the introduction of the notion of a provably correct modal truth criterion and its use in the correctness/completeness proof. The first such correctness argument is given on [55, p. 100]; although it it is very loose, my proof clearly descends from it. A series of five papers, [67,66,41,25,36], building on each other, rigorously prove correctness of linear planners. Those papers were motivated by many of the same considerations as this one: rigor requires simplicity, guarantees agreement about details, can unveil problems and suggest solutions. Thus these papers form the neat part of the scruffy-neat research cycle for linear planning. The neat part of the cycle for nonlinear planning begins with this report .

Yet I wonder about the psychological reality of this sort of planning. It may be that the only solutions to the frame problem we can devise are heuristic. Anecdotal evidence suggests that humans solve problems by improvisation, doing something easy and debugging the result when it fails [53]. Sussman's HACKER worked that way; unfortunately the set of bugs that it could patch are ones that TWEAK never introduces, and so his specific debugging techniques are of no use. [1] describes the beginning of research on improvisation.

That any problem can be viewed as a planning problem (a consequence of the undecidability theorems) suggests the slogan " 'planning' means 'computation'." It makes no more sense to talk about "the planning problem" than it does to talk about "the computation problem."

Appendix A

Proofs

A.1 The modal truth criterion

I prove the criterion in three steps. First, I prove the *time's arrow lemma*, which says that only the steps executed before a situation are relevant to what is true in that situation. Time does not flow backward. I use the time's arrow lemma to prove a truth criterion for complete plans that is analogous to the modal truth criterion. I use that and a series of lemmas about consistent sets of constraints to prove the modal truth criterion. All these proofs except the last are by numerical induction. The proof of the modal truth criterion is more interesting; in it I construct specific completions of plans that satisfy various conditions.

Time's arrow lemma: Let $\mathcal{P}$ and $\mathcal{Q}$ be complete plans whose initial situation and first n steps are identical. A proposition p is true in the initial situation or the input or output situation of one of the first n steps of $\mathcal{P}$ iff it is true in the corresponding situation in $\mathcal{Q}$.

Proof: By induction on n. If $\mathcal{P}$ and $\mathcal{Q}$ have no steps, they have only one situation, which is both initial and final, and the same in both. Certainly p is true in this situation in $\mathcal{P}$ iff it is true in the corresponding situation in $\mathcal{Q}$. Suppose now that the lemma is true for plans whose initial situation and first $n-1$ steps are identical; I will show that it holds for plans whose initial situation and first n steps are identical. Let $\mathcal{P}$ and $\mathcal{Q}$ be such plans; they have the first $n-1$ steps identical, and by the induction hypothesis p is true in the initial situation and the input and output situations of the first $n-1$ situations of $\mathcal{P}$ iff p is true in the corresponding situation of $\mathcal{Q}$. The only remaining situation we need check is the output situation s of the nth step S. By definition, s is the input situation of S minus any propositions denied by S, plus any propositions asserted by S. If p is neither asserted nor denied by S, then it is true in s just in case it is true in the input situation of S, which, by the induction hypothesis, is iff it is true in the input situation of the analogous step of $\mathcal{Q}$. If p is asserted or denied by S, it is also asserted or denied by the analogous step in $\mathcal{Q}$ and so again is true in s iff it is true

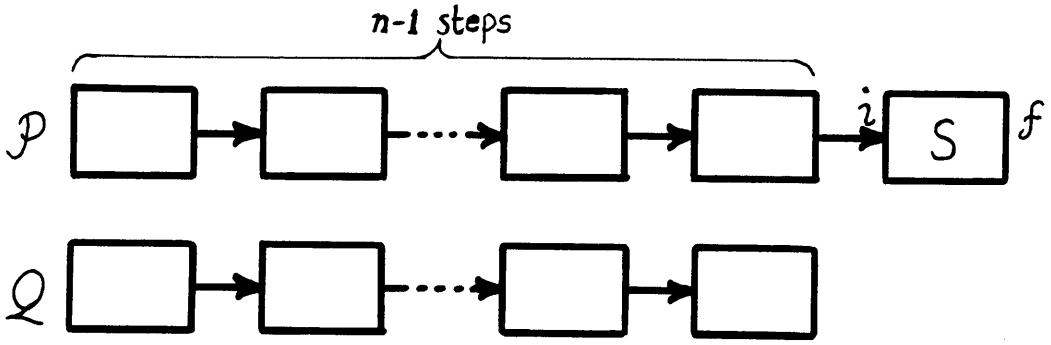
in the output situation of the nth step of $\mathcal{Q}$. By induction, then, the lemma holds for any n. □

Truth criterion for complete plans: In a complete plan, a proposition p is true in a situation s iff there exists a situation t previous or equal to s in which p is asserted and such that there is no step between t and s which denies p.

Proof: It should be obvious that this criterion is correct. Informally, we start with the initial situation and at each step delete from the set of propositions representing the world what is denied by the step and add what is asserted by it. Everything else is preserved untouched.

A rigorous proof again uses induction on the length of plans. A plan with no steps has only an initial and a final situation, and the two contain the same set of propositions. A proposition is true in the initial situation iff it codesignates with something in the initial situation, in which case it is asserted there. A proposition is true in the final situation iff it is true in the initial situation. In both cases, there is no possibility of an intervening denying step.

Suppose now that the criterion is correct for complete plans of length $n-1$; I will show that it is correct for complete plans of length n. Let $\mathcal{P}$ be a plan of length n, f the final situation of $\mathcal{P}$, S the last step in $\mathcal{P}$, i be the input situation of S, and $\mathcal{Q}$ the plan of length $n-1$ derived by removing S from $\mathcal{P}$. p is true in a situation in $\mathcal{P}$ other than f just in case p is true in the corresponding situation of $\mathcal{Q}$, by the time's arrow lemma. Then by the induction hypothesis, the criterion holds for every proposition and every situation in $\mathcal{P}$ except perhaps f (which contains the same propositions as the output situation of S). f is by definition i minus things denied by S plus whatever is asserted by S. Thus p is true in f iff it is asserted by S or true in i and not denied by S. In the former case, p is asserted in f and no step can intervene, so the criterion holds. In the latter, by induction there is a situation t in which p is asserted and no step between t and i which denies p, and since S does not deny p, there is also no step between t and f that denies p. This same t is before f, so the criterion is satisfied. The converse is trivial: if the criterion holds, either p is asserted in f or it is true in i and not denied by S. □



Constructions used in the proof of the truth criterion for complete plans

I will next state and give only proof sketches for four technical lemmas needed in the proof of the modal truth criterion.

Completion lemma: Every incomplete plan has a completion.

Proof: The time order and the codesignation relation are independent and can be considered separately. The time order can always be completed because it is just a partial order on steps and every partial order can be extended to a total order. The codesignation relation is complete when each of the finitely many variables mentioned in pre- and postconditions codesignates with a constant. A variable that does not already codesignate with a constant can only be constrained not to codesignate with finitely many constants. Since infinitely many constants exist, it's always possible to find a constant to constrain to codesignate with the variable and so complete the plan. □

Temporal consistency lemma: If s is a situation and $\{t_i\}$ is a finite set of situations such that for each i, s is possibly before t_i, then possibly s is before all the t_i.

Proof: By induction on the size of $\{t_i\}$. The main step is to see that constraining $s \prec t_j$ for some j leaves all the other $s \prec t_i$ possible. For $s \prec t_j$ to make $s \prec t_i$ impossible, it would have to make $t_i \prec s$ necessary. But, from the definition of the time order, from $s \prec t_j$ together with the previous constraints, there follow only constraints of the form $a \prec b$, where $a \preceq s$ and $t_j \preceq b$. □

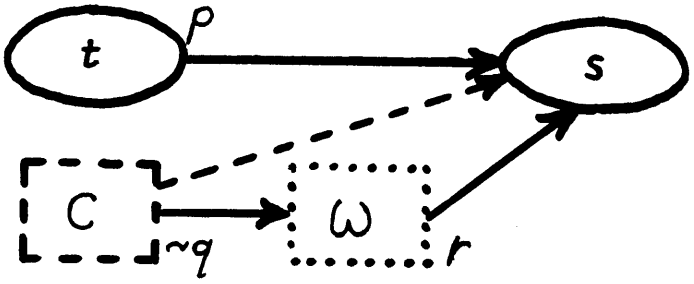
Codesignation consistency lemma: If $\{v_i\}$ is a finite set of variables none of which is constrained to codesignate with any constant, you can assign to each equivalence class (under the codesignation relation) a distinct "gensymmed" constant not previously appearing in the plan, and constrain codesignation of these constants with the variables in their respective classes.

Proof: There are enough constants because only finitely many of the infinite set can be mentioned in a plan. Since the gensymmed constants are not mentioned in the plan, they can not have been constrained to not codesignate with any of the variables. Since the $\{v_i\}$ do not codesignate with any constants, there is no problem of distinct constants being constrained to codesignate. □

Noncodesignation consistency lemma: If p is a proposition and $\{q_i\}$ is a finite set of propositions such that for all i possibly $q_i \not\approx p$ then possibly for all i $q_i \not\approx p$.

Proof: Again by induction, on the size of $\{q_i\}$. p and q_j are constrained to not codesignate by constraining some pair of elements to not codesignate. I must show that doing so does not make p and some q_i necessarily codesignate. This is because the definition of the codesignation relation is such that a codesignation of elements can not be made to follow by adding a noncodesignation. □

Modal truth criterion: A proposition p is necessarily true in a situation s iff two conditions hold: there is a situation t equal or necessarily previous to s in which p is necessarily asserted; and for every step C possibly before s and every proposition q possibly codesignating with p which C denies, there is a step W necessarily between C and s which asserts r, a proposition such that r and p codesignate whenever p and q codesignate. The criterion for possible truth is exactly analogous, with all the modalities switched (read "necessary" for "possible" and vice versa).



The necessary truth criterion. Solid lines indicate necessarily time-relatedness and dashed lines possible time-relatedness; the dashed box, a disallowed step; the dotted box a step that would make the dashed step legal.

Proof: I'll give the proof for necessary truth; the proof for possible truth is analogous. David McAllester helped debug and simplify this proof.

The criterion is composed of two independent conjuncts: existence of an establisher and absence of a clobberer. We can distinguish four cases according to whether or not

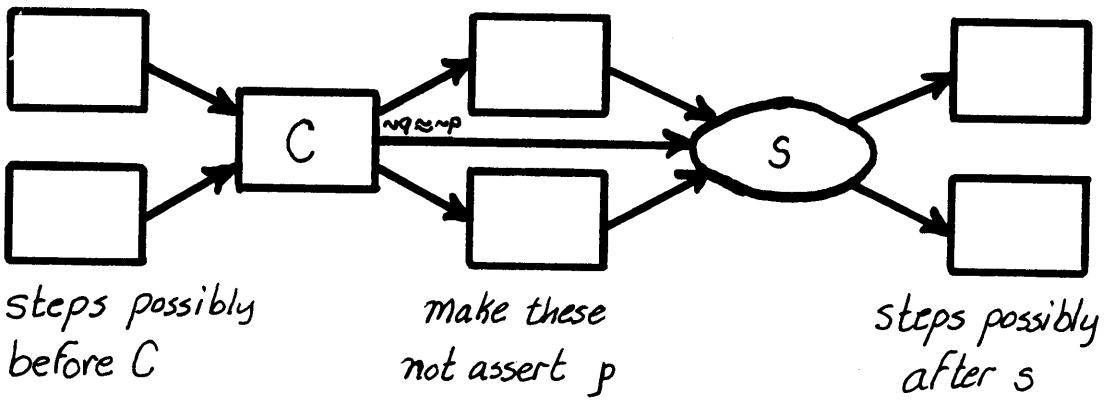
there is an establisher and whether or not there is a clobberer. To prove the criterion, I need to show that in all the cases except that in which there is an establisher and no clobberer, p is not necessarily true in s. This demonstrates the equivalence. The basic proof technique is constructive: for each of these cases, I show how to construct a specific completion of the given incomplete plan in which the truth criterion for complete plans can be used to show that p is (or is not) true in s.

I will lump together two of the four cases, those in which there is no establisher. To construct the completion falsifying p in s, first add constraints that put after s every step possibly after s. This can be done, by the temporal consistency lemma.

The next step is to apply the codesignation consistency lemma and constrain every equivalence class of variables that is not already constrained to codesignate with some specific constant to codesignate with a distinct "gensymmed" constant.

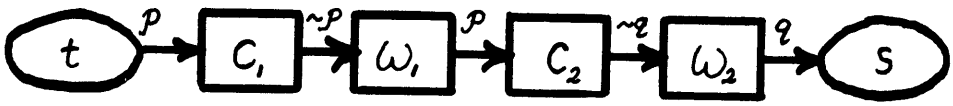
A completion in which p is not true in s is made by taking any completion of this modified plan. A completion exists, by the completion lemma. In the original incomplete plan there was no step that necessarily asserts p and that is necessarily before s; and any step that only possibly asserted p has been made not to assert it (but rather some other proposition involving gensymmed constants); and any step that necessarily asserts p but is only possibly before s has been put after s. So no situation before or equal to s asserts p, and by the truth criterion for complete plans p is not true in s in this completion.

The next case is that in which there is a clobberer C. To construct the falsifying completion first constrain C before s, which is possible because C is a clobberer. Then constrain every step still possibly after s to be actually after s, every step still possibly before C to be actually before C. This can be done, by application of the temporal consistency lemma, twice. Now constrain codesignation of q (the proposition possibly codesignating with p which C denies) and p, which can be done because C is a clobberer, and constrain noncodesignation with p of every postcondition of every step between C and s, which can be done by the noncodesignation consistency lemma and the observation that any such step, if it now necessarily asserted p, would be a white knight. Finally arbitrarily complete the result (using the completion lemma). By the truth criterion for complete plans, in this completion p is false in s. C is a step that denies p, it is before s, and no step in-between asserts p.



Falsifying p in a plan with a clobberer

The last case is that in which there is an establisher and no clobberer. In this case, I will show, p is necessarily true in s. Choose any completion. Since there is an establisher t, p is true in t. Consider the set of steps $\{C_i\}$ that possibly denied p in the incomplete plan. Each of these either does or does not actually deny p. The latter sort we can ignore; they do not endanger p's truth in s. Since the remaining C_i are not clobberers, there is for each a corresponding white knight W_i asserting p (since the white knight asserts p whenever C_i denies it). There may in turn be a step after the white knight denying p; but it must also have its own white knight. Since there are only finitely many steps in the plan, eventually p will be asserted by a white knight and not denied before s. Then by the truth criterion for complete plans, p is true in s. □



A completion of a plan satisfying the criterion.

A.2 The outcomes lemma

Outcomes lemma: Each of the three possible outcomes of TWEAK's algorithm (success, failure, and looping) is possible for some choice of domain and problem.

Proof: A trivial example of success is a problem with a single goal which is true in the initial situation. A trivial example of failure is provided by a problem that has at least one goal that is not true in the initial situation and which is not possibly asserted

by any available step template. An example of non-termination is given by the problem whose initial state is $\sim g$ and $\sim h$ and whose goals are g and h in a domain in which there are two step templates, one with precondition $\sim h$ and postcondition g and the other with precondition $\sim g$ and postcondition h. TWEAK loops on this problem, building plans that are longer and longer chains of steps that alternately assert g and h. □

A.3 The correctness/completeness theorem

Correctness/completeness theorem: If TWEAK, given a problem, terminates claiming a solution, the plan it produces does in fact solve the problem. If TWEAK returns signalling failure or does not halt, no solution exists.

Proof: This follows directly from the use of the necessary truth criterion in computing whether a plan solves the problem given and in constructing the goal achievement procedure. TWEAK's incomplete plan always has the same initial situation as the problem given, and the top-level loop continues until all the goals are achieved, at which point the plan must solve the problem. If a solution exists it must be a plan that in some way achieves the problem's goals. Since TWEAK's search is breadth-first, and since the nondeterministic plan modification procedure generates only finitely many ways to produce a new plan from an old one, TWEAK eventually examines every way to satisfy a problem's goals. The plan must also have all preconditions achieved; but TWEAK also tries all ways to achieve preconditions. Thus, if a solution exists, TWEAK will find it. □

A.4 First undecidability theorem

First undecidability theorem: Any Turing machine with its input can be encoded as a planning problem in the TWEAK representation. Therefore, planning is undecidable, and no upper bound can be put on the amount of time required to solve a problem.

Proof: The encoding is direct and straightforward. An infinite set of constants t_i are used to represent the tape squares. The binary relation successor represents the connectivity of the tape. The functional binary relation contents represents the contents of the tape, and the set of constants a_i represent the alphabet written on the tape. A predicate head holds of exactly one tape square, that under the head. A set of constants s_i represent the finitely many states of the controller, and the predicate state holds of the current state only. For each arc from state s_i to state s_j in the controller's state graph there is an operator type. The operator has four preconditions: (state s_i), (head t) where t is a variable representing the unknown square under the head, (contents t a), where a is the symbol the arc specifies to read, and (successor t u) if the arc says

to move right, or (successor u t) if the arc says to move left. The operator has four postconditions: (state s_j), ~(state i), (head u), (contents t b), where b is the written symbol, and ~(contents t a).

The initial situation of the problem has (state i), where i is the initial state of the Turing machine controller, and (head t), where t is the initial tape square. The input to the machine is specified via the contents relation. There must be countably many successor propositions to encode the topology of the tape (and also countably many contents propositions to make all but finitely many squares blank). The final situation of the problem is just (state f), where f represents the halt state for the Turing machine.

It is easy to see that a valid plan for this problem amounts to a trace of the encoded Turing machine computation. Such a plan exists iff the Turing machine halts. Thus it is undecidable whether or not there exists a plan. □

A.5 The intractability theorem

Intractability theorem: The problem of determining whether a proposition is necessarily true in a nonlinear plan whose action representation is sufficiently strong to represent conditional actions, dependency of effects on input situations, or derived side effects is NP-hard.

Proof: This proof is based on an idea of Stan Rosenschein's. It is by direct reduction from PSAT, or rather from the equivalent problem of determining whether a boolean formula is valid (true under every truth assignment).

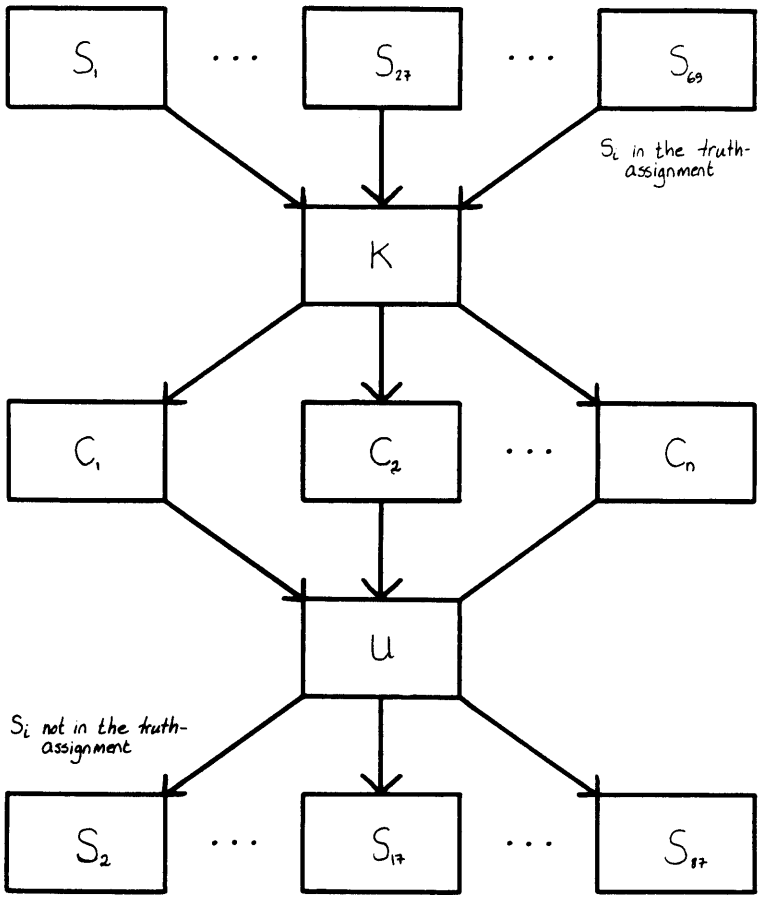
Augment TWEAK's action representation by adding a new type of step, the conditional step. A conditional step is always applicable, but has two sets of postconditions, the if-true and the if-false postconditions. The if-true postconditions hold in the output situation if all the preconditions were satisfied in the input situation; otherwise the if-false postconditions hold.

Given a propositional formula p on atoms p_i, I construct a plan in this representation and a proposition such that the proposition is necessarily true in the final situation iff p is valid. The p_i are recycled as propositions in the plan, and all are made false in the initial situation. A set of steps S_i make the p_i true, so that a truth assignment on the p_i corresponds to a subset of $\{S_i\}$.

A set of conditional steps, $\{C_j\}$, check that p holds in every truth assignment. Let p be expressed as the disjunction of conjunctive clauses. There is a C_j for each clause; C_j has as its preconditions the conjunct p_i in the corresponding clause. The if-true postconditions assert a proposition called *satisfied*; the if-false postconditions are null. *satisfied* is false in the initial situation.

What remains is to guarantee that all the C_j evaluate their clauses relative to the same truth assignment. This is done by introducing a new "flag" proposition called *checking*, initially false, which is made true by a single step K and false by a single step

U. K and U have no preconditions and the are ordered respectively before and after all the C_j. The S_i are made conditional, so that they assert *satisfied* if *checking* holds.



When does *satisfied* hold in the final situation? It holds whenever there are any S_i between K and U, by the mechanism of the last paragraph. If there are no S_i between K and U, the C_j evaluate their clauses in the truth assignment corresponding to the set of S_i before K. (The remaining S_i are after U and have no effect on *satisfied*.) Since every truth assignment is generated by some completion of the plan, *satisfied* is necessarily true in the final situation just in case p is in fact valid.

This establishes the result for conditional actions. For the rest, it is easy to see that these conditionals can be simulated with dependency of effects on input situations or derived side effects. □

A.6 Second undecidability theorem

Second undecidability theorem: Planning is undecidable even with a finite initial situation if the action representation is extended to represent actions whose effects are a function of their input situation.

Proof: Papert and McNaughton [34] showed that any recursive function can be computed by a two-counter machine, i.e. a machine consisting of two positive-integer-valued registers and a finite-state control which can test either register for equality to zero and can increment or decrement either register. I encode such a machine, with inputs, as a planning problem, the goal of which is to get the machine into the halt state. I use three unary predicates to represent the state of the machine, counter1, counter2, and state. States and integers are represented by constants. counter1 and counter2 hold of exactly one integer and represent the contents of the two counters; state holds of exactly one state of the finite state controller and represents the state the controller is in.

The initial situation of the problem has (state s) where s is the start state and (counter1 c_1) and (counter2 c_2) where c_1 and c_2 are the values the two-counter machine is started with. There is an operator for each arc in the finite state controller. The operator associated with the arc (s_1, s_2) has as a precondition (state s_1) and as postconditions (state s_2) and $\sim$(state s_1). The operators associated with increment and decrement arcs also have increment and decrements of counter1and counter2as appropriate. The branches are implemented in the finite state machine with nodes that have two arcs coming out of them, one labeled $c_i > 0$ and one labeled $c_i = 0$ (for i in $\{1, 2\}$). These correspond to operators that have those same tests as preconditions and no postconditions other than setting state.

Now any plan that solves this problem is a trace of the computation that would be executed by the two-counter machine. The planner has to do at least as much work as the simulated machine did. □

Appendix B

Code

This appendix gives the code for TWEAK's top-level control structure and nondeterministic achievement procedure. This is the entire planner except for the plan representation and constraint maintenance. These are long and messy, but completely straightforward. The code given here makes no attempt whatsoever to be efficient; many obvious optimizations are possible. I've omitted them in order to make the structure of the program transparent.

The code is written in Dependency-Directed Lisp. This has the syntax and semantics of Common Lisp, except for the dependency-directed choice construct `choose`. `choose` takes an arbitrary number of subforms, nondeterministically executes one of them, and returns that as its value. Written in terms of `choose` is `choose-member`, which, given a list, returns an arbitrary member of the list. It can be defined as follows:

```
(defun choose-member (l)
  (cond ((null l) (fail))
        (t (choose (first l) (choose-member (rest l))))))
```

`fail` is the special form that invokes dependency-directed backtracking.

```
;;; The current incomplete plan -- a set of steps -- lives here.
(defvar *steps*)

;;; The top-level loop of the planner.
;;; Keeps achieving goals until none are left; then returns the plan.
(defun planner-loop ()
  (let ((unachieved-goals (unachieved-goals)))
    (cond ((null unachieved-goals)
           ;; we're done, return the plan
           *steps*)
          (t
           ;; choose a goal to achieve
           (let ((goal (choose-member unachieved-goals)))
             ;; achieve it
             (achieve (goal-proposition goal)
                      (goal-situation goal)))
           ;; achieve some more goals.
           (planner-loop))))

;;; Achieve proposition p in situation s.
(defun achieve (p s)
  (establish p s)
  (declobber p s))

(defun establish (p s)
  ;; t1 is what's called t in the text;
  ;; the variable t is sacred to Lisp.
  (let ((t1 (choose
             ;; simple establishment -- try it first
             (choose-member *situations*)
             ;; establishment by step addition
             (let ((new-step (choose-member
                              (steps-from-means-ends-analysis p))))
               (add-step-to-plan new-step)
               (step-output-situation new-step)))))
    (cond ((not (eq t1 s))
           (constrain-before t1 s)))
    (constrain-propositions-codesignating
      p
      (choose-member
        (propositions-necessarily-asserted-in-situation t1)))))
```

```
;;; Make sure proposition p is not clobbered in situation s.
(defun declobber (p s)
  (let ((clobberers (clobberers p s)))
    ;; if no clobberers, we are done
    (cond ((not (null clobberers))
           (let ((c (first clobberers)))
             (choose (constrain-before s c)      ; promotion
                     (dolist (q (step-postconditions c))
                       (choose
                         (constrain-propositions-not-codesignating
                           q (negation p))       ; separation
                         (declobber-by-white-knight p q s c)))))
             ;; now do it again to get rid of the rest of the clobberers
           (declobber p s)))))

(defun declobber-by-white-knight (p q s c)
  (let ((w (choose (choose-member *steps*)
                   (choose-member (steps-from-means-ends-analysis p)))))
    (constrain-before c w)
    (constrain-before w s)
    (constrain-codesignation-implication
      p q (choose-member (step-postconditions w)))))

;;; This returns a list of steps that could possibly achieve the
;;; argument.
;;;
;;; A step-template is instantiated by copying it,
;;; changing the names of the variables.
(defun steps-from-means-ends-analysis (proposition)
  (let ((steps '()))
    (dolist (template *step-templates*)
      (cond ((possibly-asserts?  template proposition)
             (push (instantiate-step-template template) steps))))
    steps))
```

```
;;; Truth of propositions in situations:  the modal truth criterion and
;;; related stuff.
;;; As well as being a predicate, this returns the actual postcondition.
(defun necessarily-asserts?  (step proposition)
  (some (lambda (postcondition)
          (necessarily-codesignating?  proposition postcondition))
        (step-postconditions step)))

(defun possibly-asserts?  (step proposition)
  (some (lambda (postcondition)
          (possibly-codesignating?  proposition postcondition))
        (step-postconditions step)))

(defun necessarily-true?  (p s)
  (and (some (lambda (t1)
               (and (or (eq t1 s) (necessarily-before?  t1 s))
                    (necessarily-asserted-in?  p t1)))
             *situations*)
       (not (some (lambda (c) (clobbers?  c p s)) *steps*))))

(defun clobbers?  (c p s)
  (and (possibly-before?  c s)
       (some (lambda (not-q)
               (let ((q (negation not-q)))
                 (and (possibly-codesignating?  p q)
                      (not (some (lambda (w)
                                   (and (necessarily-between?  w c s)
                                        (some (lambda (r)
                                                (codesignation-implied?
                                                  p q r))
                                              (step-postconditions w))))
                                 *steps*)))))
             (step-postconditions c))))
```

```
(defun necessarily-asserted-in?  (proposition situation)
  (typecase situation
    ((or final-situation input-situation)
     nil)
    (initial-situation
     (some (lambda (initial-proposition)
             (necessarily-codesignating?  proposition
                                          initial-proposition))
           (initial-situation-propositions situation)))
    (output-situation
     (necessarily-asserts?  (step-situation-step situation)
                            proposition))
    (otherwise
     (ferror "That's not a situation!"))))

(defun possibly-asserted-in?  (proposition situation)
  (typecase situation
    ((or final-situation input-situation)
     nil)
    (initial-situation
     (some (lambda (initial-proposition)
             (possibly-codesignating?  proposition initial-proposition))
           (initial-situation-propositions situation)))
    (output-situation
     (possibly-asserts?  (step-situation-step situation) proposition))
    (otherwise
     (ferror "That's not a situation!"))))

(defun propositions-necessarily-asserted-in-situation (situation)
  (typecase situation
    ((or final-situation input-situation)
     '())
    (initial-situation
     (initial-situation-propositions situation))
    (output-situation
     (step-postconditions (step-situation-step situation)))
    (otherwise
     (ferror "That's not a situation!"))))
```

```
;;; Unachieved preconditions are ones that are not necessarily true in
;;; the input situation.
(defun unachieved-preconditions (step)
  (let ((unachieved-preconditions '()))
    (dolist (precondition (step-preconditions step))
      (cond ((not (necessarily-true?  precondition
                                      (step-input-situation step)))
             (push precondition unachieved-preconditions))))
    unachieved-preconditions))

;;; A goal is a cons of a proposition and the situation in which it must
;;; be true.
(defmacro goal-proposition (goal) `(car ,goal))
(defmacro goal-situation (goal) `(cdr ,goal))
(defmacro make-goal (proposition situation)
  `(cons ,proposition ,situation))

(defun unachieved-precondition-goals (step)
  (mapcar (lambda (precondition)
            (make-goal precondition (step-input-situation step)))
          (unachieved-preconditions step)))

(defun unachieved-goals ()
  (let ((unachieved-goals '()))
    ;; goals of the problem
    (dolist (goal-proposition *top-level-goal-propositions*)
      (cond ((not (necessarily-true?  goal-proposition
                                      *final-situation*))
             (push (make-goal goal-proposition *final-situation*)
                   unachieved-goals))))
    ;; precondition goals
    (dolist (step *steps*)
      (setq unachieved-goals
            (nconc (unachieved-precondition-goals step)
                   unachieved-goals)))
    unachieved-goals))

(defun clobberers (p s)
  (let ((clobberers '()))
    (dolist (c *steps*)
      (cond ((clobbers?  c p s) (push c clobberers))))
    clobberers))
```

Bibliography

[1] Agre, Philip E., "Routines." MIT AI Memo 828, May, 1985.

[2] Allen, James F., and Koomen, Johannes A., "Planning Using a Temporal World Model." *IJCAI-83*, pp. 741–747.

[3] Batali, John, "Computational Introspection." MIT AI Memo 701, February, 1983.

[4] Chapman, David, "Naive Problem Solving and Naive Mathematics." MIT AI Working Paper 249, June, 1983.

[5] Chapman, David, "Cognitive Cliches." Forthcoming.

[6] Corkill, Daniel D., "Hierarchical Planning in a Distributed Environment." *IJCAI-79*, pp. 168–175.

[7] Davis, Randall, "Meta-Rules: Reasoning about Control." *Artificial Intelligence* 15 (1980), 179–222.

[8] Daniel, L., "Planning: Modifying Non-linear Plans." Edinburgh AI Working Paper 24, Edinburgh University, 1977. Cited in [62].

[9] deKleer, Johan, Doyle, Jon, Steele, Guy L. Jr., and Sussman, Gerald Jay, "Explicit Control of Reasoning." *ACM SIGPLAN Notices* Vol. 12, No. 8/*ACM SIGART Newsletter* No. 64, combined special issue, proceedings of the Symposium on Artificial Intelligence and Programming Languages, August 1977, pp. 116-125. Also MIT AI Memo No. 427, June 1977.

[10] deKleer, Johan, Doyle, Jon, Rich, Charles, Steele, Guy L. Jr., and Sussman, Gerald Jay, "AMORD, a Deductive Procedure System." MIT AI Memo 435, January, 1978.

[11] Doyle, Jon, *Truth Maintenance Systems For Problem Solving.* MIT AI Technical Report 419, January 1978.

[12] Doyle, Jon, *A Model for Deliberation, Action, and Introspection.* MIT AI Technical Report 581, Cambridge Mass., May, 1980.

[13] Fahlman, Scott Elliott, "A Planning System for Robot Construction Tasks." *Artificial Intelligence* 5 (1974) pp. 1–49.

[14] Fikes, Richard E., and Nilsson, Nils J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *Artificial Intelligence* **2** (1971), pp. 198–208.

[15] Fikes, Richard E., Hart, P.E., and Nilsson, Nils J., "Some New Directions in Robot Problem Solving." Chapter 23 in *Machine Intelligence 7*, Meltzer and Mitchie, eds., Edinburgh University Press, Edinburgh 1972.

[16] Georgeff, Michael, "A Theory of Action for MultiAgent Planning." *AAAI-84*, pp. 121–125.

[17] Garey, Michael R., and Johnson, David S., *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[18] Hammond, Kristian J., "Planning and Goal Interaction: The use of past solutions in present situations." *AAAI-83*, pp. 148–151.

[19] Hayes, Philip J., "A Representation For Robot Plans." *4th IJCAI.*

[20] Hayes-Roth, Barbara, Hayes-Roth, Frederic, Rosenschein, Stan, and Cammarata, Stephanie, "Modeling Planning as an Incremental, Opportunistic Process." *IJCAI-79*, pp. 375–383.

[21] Hewitt, Carl, "Procedural Embedding of Knowledge in PLANNER." *IJCAI-71*, pp. 167–182.

[22] Hewitt, Carl, *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot.* MIT AI Technical Report 258, April, 1972.

[23] JanLert, Lars-Erik, "Modeling Change—the Frame Problem." To appear in *The Frame Problem and Other Problems of Holism in Artificial Intelligence*, Zenon Pylyshyn, ed., Ablex Publishing, 1985.

[24] "Planning within first-order dynamic logic." *Proceedings of the Fourth Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI)*, Saskatoon, May 1982.

[25] *A First Order Dynamic Logic for Planning.* Tech. Report CSRG-144, Department of Computer Science, University of Toronto, May 1982.

[26] Kibler, Dennis, and Morris, Paul, "Don't be Stupid." *IJCAI-81.*

[27] Lansky, Amy, "Behavioral Planning for Multi-Agent Domains." Technical Note, SRI AI Center, forthcoming.

[28] London, Phil, "A Dependency-Based Modeling Mechanism for Problem Solving." Computer Science Technical Report 589, University of Maryland, College Park, Maryland, November 1977.

[29] London, Phil, "Dependency Networks as a Representation for Modeling in General Problem Solvers." Computer Science Technical Report 698, University of Maryland, College Park, Maryland, September 1978.

[30] McCarthy, J., and Hayes, P. J., "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in *Machine Intelligence 4*, ed. by B. Meltzer and D. Mitchie, Edinburgh, pp. 463–502.

[31] McDermott, Drew. "Generalizing Problem Reduction: A Logical Analysis." *IJCAI-83.*

[32] Miller, David, Firby, R. James, and Dean, Thomas, "Deadlines, Travel Time, and Robot Problem Solving." Submitted to *IJCAI-85.*

[33] Milne, A.A., *Winnie The Pooh.* Dell Publishing Company, New York, 1984. First copyright 1926.

[34] Minsky, Marvin L., *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

[35] Newell, A., Shaw, J.C., and Simon, H.A., "Report on a general problem-solving program." *Proceedings of the International Conference on Information Processing*, pp. 256–264, UNESCO, Paris 1960. Reprinted in *Computers and Automation*, July 1959.

[36] Pednault, Edwin P. D., "Preliminary Report on a Theory of Plan Synthesis." SRI AI Center Technical Note 358, September, 1985.

[37] Raphael, Bertram, "The Frame Problem in Problem-Solving Systems." *Proceedings of the Advanced Study Institute on Artificial Intelligence and Heuristic Programming*, Menaggio, Italy, 1970.

[38] Rhys, J., "A selection problem of shared fixed costs and network flows." *Management Science*, Vol. 17, 1970.

[39] Rich, Charles, *Inspection Methods in Programming.* MIT AI Technical Report 604, Cambridge Mass., June, 1981.

[40] Rich, Charles, "A Formal Representation for Plans in the Programmer's Apprentice." *IJCAI-81*, pp. 1044–1052.

[41] Rosenschein, Stanley J., "Plan Synthesis: A Logical Perspective." *IJCAI-81*, pp. 331–337.

[42] Rosenschein, Jeffrey S., "Synchronization of Multi-Agent Plans." *AAAI-82*, pp. 115–119.

[43] Rosenschein, Stanley J., "Formal Theories of Knowledge in AI and Robotics." SRI AI Center Technical Note 362, September 10, 1985.

[44] Sacerdoti, Earl D., "The Nonlinear Nature of Plans." *Advance Papers of the 4th IJCAI*, 1975, pp. 206–214.

[45] Sacerdoti, Earl D., *A Structure for Plans and Behavior*. American Elsevier, New York, 1977. Also SRI AI Technical Note 109, August, 1975.

[46] Sacerdoti, Earl D., "Problem Solving Tactics." *Proceedings of the 6th IJCAI*, 1979, pp. 1077–1085.

[47] Siklossy, L., and Roach, J., "Collaborative Problem-Solving Between Optimistic and Pessimistic Problem Solvers." *IFIP-74*.

[48] Stallman, Richard M., and Sussman, Gerald Jay, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis." MIT AI Memo 380, September 1976.

[49] Stefik, Mark Jeffrey, *Planning with Constraints*. PhD thesis, Stanford University, January 1980. Also Stanford Heuristic Programming Project Memo 80-2 and Stanford Computer Science Department Memo 80-784.

[50] Stefik, Mark, "Planning and Metaplanning (MOLGEN: Part 2)." *Artificial Intelligence* **16** (1981) pp. 141–170.

[51] Shrobe, Howard Elliot, *Dependency Directed Reasoning for Complex Program Understanding*. MIT AI Technical Report 503, April, 1979.

[52] Shrobe, Howard Elliot, "Dependency Directed Reasoning in the Analysis of Programs Which Modify Complex Data Structures." *Proceedings of the Sixth IJCAI*, 1979, pp. 829–835.

[53] Suchman, Lucy A., *Plans and Situated Actions: The problem of human-machine communication*. Xerox Palo Alto Research Center, Palo Alto, 1985.

[54] Sussman, Gerald Jay, and McDermott, Drew Vincent, "From PLANNER to CONNIVER—A genetic approach." *Proceedings of the Fall Joint Computer Conference*, 1972, pp. 1171-1179.

[55] Sussman, Gerald Jay, *A Computational Model of Skill Acquisition*. MIT AI Technical Report 297, August 1973. Also American Elsevier, New York, 1975.

[56] Tate, Austin, "INTERPLAN: A plan generation system which can deal with interactions between goals." Machine Intelligence Research Unit Memorandum MIP-R-109, University of Edinburgh, Edinburgh, December 1974.

[57] Tate, Austin, "Interacting Goals and Their Use." *Advance Papers of the 4th IJCAI*, 1975.

[58] Tate, Brian Austin, *Using Goal Structure to Direct Search in a Problem Solver.* PhD thesis, University of Edinburgh, 1975.

[59] Tate, Austin, "Project Planning Using a Hierarchic Nonlinear Planner." Department of Artificial Intelligence Research Report No. 25, University of Edinburgh, Edinburgh, August 1976.

[60] Tate, Austin, "Generating Project Networks." *Proceedings of the 5th IJCAI*, 1977.

[61] Tate, Austin, "Planning in Expert Systems". Invited paper for the *Alvey IKBS Expert Systems Theme—First Workshop* at Cosener's House, Abingdon, Oxford, 3–5 March, 1984. Also D.A.I. Research Paper 221, University of Edinburgh.

[62] Tate, Austin, "Planning and Condition Monitoring in a FMS." *Proceedings of the International Conference on Flexible Manufacturing Systems*, London, July 1984.

[63] Tate, Austin, "Goal Structure—Capturing the Intent of Plans." *ECAI-84: Advances in Artificial Intelligence*, T. O'Shea, ed., Elsevier Science Publications B.V. (North-Holland), 1984.

[64] Van Baalen, Jeffrey, "Planning and Exception Handling." Forthcoming.

[65] Vere, Steven A., "Planning In Time: Windows and Durations for Activities and Goals." *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. PAMI-5, No. 3, May 1983. pp 246–267.

[66] Waldinger, Richard, "Achieving Several Goals Simultaneously." SRI Artificial Intelligence Center Technical Note 107, Menlo Park, July 1975.

[67] Warren, David H. D., "WARPLAN: A System For Generating Plans." Department of Computational Logic Memo No. 76, University of Edinburgh, Edinburgh, June 1974.

[68] Warren, David H. D., "Generating Conditional Plans and Programs." *Proceedings of the AISB Summer Conference*, University of Edinburgh, July 1976, pp. 344–354.

[69] Wilensky, Robert, "Meta-Planning: Representing and Using Knowledge About Planning in Problem Solving and Natural Language Understanding." *Cognitive Science* 5 (1981), pp. 197–233.

[70] Wilensky, Robert, *Planning and Understanding: A Computational Approach to Human Reasoning.* Reading, Massachusetts: Addison-Wesley, 1983.

[71] Wilkins, David E., "Representation in a Domain-Independent Planner." *IJCAI-83.*

[72] Wilkins, David E., "Domain-Independent Planning: Representation and Plan Generation." *Artificial Intelligence* 22:3 (1984) pp. 269–301. Also SRI International Technical Note No. 266R, Menlo Park, California, May 5, 1983.

[73] Wilkins, David E., "Recovering from execution errors in SIPE." *Computational Intelligence* **1**, pp. 33-45 (1985).

Index

AMORD 36
assertion 6
before 10
block promotion 29
clobberer 9
code, for TWEAK 48
codesignation consistency lemma 41
codesignation constraint 5
cognitive cliche 24
compacting 30
complete plan 5
completeness 13
completion 5
completion lemma 40
CONNIVER 36
constant 5
constraint posting 4
consumer operator 24
content 5
correctness/completeness theorem 13
    proof of 44
DDL 36
declobbering 29
demotion 29
denial 6
dependency-directed backtracking 34
Dependency-Directed Lisp 36
dependent operator 24
element 5
establisher 9
establishment, simple 29
final situation 6, 10
first undecidability theorem 13
    proof of 44
FORBIN 25
frame problem 23
goal 10
goal-achievement procedure 10
GPS 20
HACKER 20
incomplete plan 5
individual promotion 29
initial situation 6, 10
input situation 6
instantiation of steps 12
intermediate technique 24
INTERPLAN 20
intractability theorem 23
    proof of 45
metaplanning 35
modal truth criterion 9
    proof of 41
MOLGEN 20
necessary 5
negated 5
negation 5
NOAH 20
noncodesignation consistency lemma 41
NONLIN 20