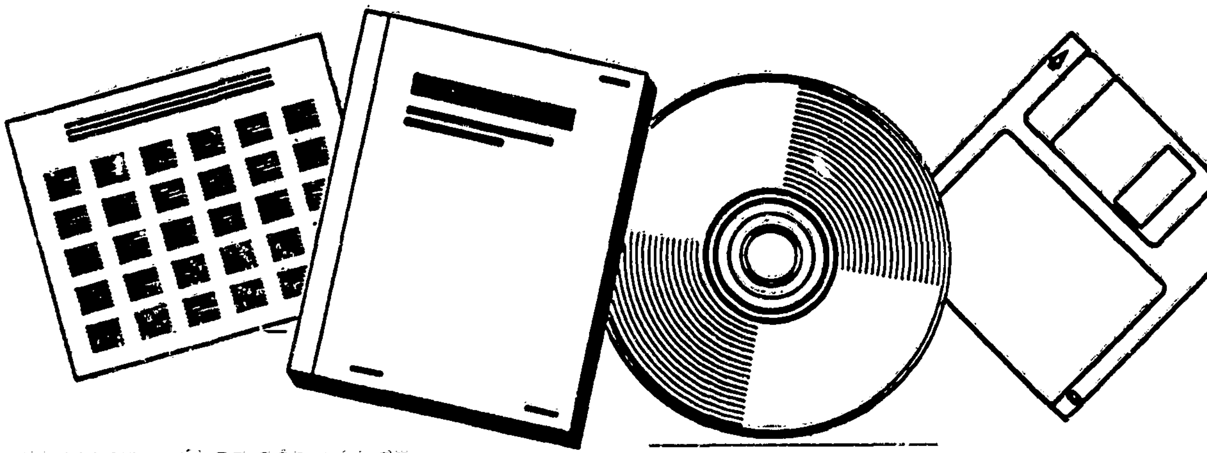
AUTOMATIC ABSTRACTION IN PLANNING

STANFORD UNIV., CA

MAR 91



March 1991

Report No. STAN-CS-91-1357

Thesis

PB96-149711

AUTOMATIC ABSTRACTION IN PLANNING

by

Jens Christensen

Department of Computer Science

Stanford University

Stanford, California 94305

AUTOMATIC ABSTRACTION IN PLANNING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Jens Christensen
March 1991

Abstract

Traditionally, abstraction in planning has been accomplished by either state abstraction or operator abstraction, neither of which has been fully automatic. We present a new method, *predicate relaxation*, for *automatically* performing state abstraction. Predicate relaxation generates abstraction hierarchies that, for some domains, can be more useful than those generated by previous abstraction mechanisms. PABLO, a nonlinear hierarchical planner, implements predicate relaxation. Theoretical, as well as empirical results are presented which demonstrate the potential advantages of using predicate relaxation in planning. Relaxed predicates can also be used by PABLO to achieve a limited form of reactivity, whereby an executable sequence of actions is constructed in case of interruption.

We also present a new definition of hierarchical operators that allows us to guarantee a limited form of completeness. This new definition is shown to be, in some ways, more flexible than previous definitions of hierarchical operators. The ability to plan using such operators has been incorporated into PABLO.

Finally, a *Classical Truth Criterion* is presented that is proven to be sound and complete for a planning formalism that is general enough to include most classical planning formalisms that are based on the STRIPS assumption.

Acknowledgements

First and foremost I would like to thank Neguine, whom I met, courted, and married during the five years I spent in the Ph.D. program at Stanford. She helped make the experience a pleasant and memorable one.

The research itself would have been impossible without the enlightened guidance of Nils Nilsson whose encouragement and insights became indispensable. I would also like to thank my other two advisers, David Wilkins and Yoav Shoham, who brought different viewpoints to the problems I was tackling, and made many thoughtful contributions along the way.

The Principia group provided an excellent forum for airing ideas. Special thanks go to Adam Grove, without whose help the chapter on the Classical Truth Criterion could not have been written. Andrew Kosoresow, Karen Myers, Rich Washington, Eunok Paek, and Alon Levy reviewed the work and made valuable suggestions. Matt Ginsberg also made many useful suggestions along the way.

No acknowledgements would be complete without mentioning my parents whose insistence on my coming to Stanford as well as their support and encouragement along the way are a major reason for the successful completion of this thesis.

Finally, I would like to acknowledge the support of NASA through NASA Grant NCC2-494 and Texas Instruments through Contract No. 7554900.

Contents

List of Tables

List of Figures

Chapter 1

Review of Abstraction in Planning

One of the powerful tools employed by planners to deal with the complexity of planning problems is abstraction. Although widely used, and in many guises, abstraction remains relatively poorly understood. Because of this, systems employing abstraction have usually left the definition of the abstractions up to the user of the system. This thesis introduces a new method for performing abstractions *automatically* and presents results related to abstraction in planning which should help to clarify the potential benefits, as well as drawbacks, of using abstraction in planning. In this chapter, we present the different methods of abstraction employed up until now, pointing out some potential problems along the way.

The rest of the thesis is organized as follows. Chapter 2 introduces *predicate relaxation,* a new method for performing automatic abstraction in planning. Chapter 3 presents PABLO, a nonlinear planner that implements predicate relaxation. Chapter 4 discusses theoretical results pertaining to the complexity of using predicate relaxation in planning. Chapter 5 presents empirical results demonstrating the increased efficiency gained when using predicate relaxation in planning. Chapter 6 discusses how an extension to predicate relaxation can be used to achieve a limited form of reactivity in planning. Chapter 7 discusses a method for performing operator hierarchicalization in planning, which guarantees a limited form of completeness. Chapter 8 describes how both abstraction techniques can be used effectively in planning. Chapter 9 describes a new truth criterion for planning which is based on a very general planning formalism. Finally, chapter 10 discusses open problems and further work that needs to be done in the area of abstraction in planning.

1.1 Basic Concepts

When thinking about abstraction it is often useful to do so in the context of a state space graph. In such a graph each node corresponds to a particular world state, and each directed arc to a particular operator which transforms that world state into another. Figure 1.1 is an example of a general state space graph.

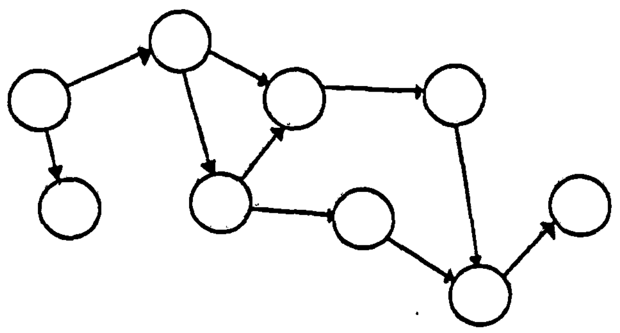


Figure 1.1: State space graph

It is the task of a planner, when given a description of a particular *initial state* in the state space graph as well as a description of one or more desired *goal states*, to discover one or more paths from the initial state to one of the goal states. Of course the state space graph is often prohibitively large, possibly infinite, and is therefore not usually explicitly represented. Rather, the state space graph is implicitly defined by a set of operators, i.e. functions from states to states. Parts of the state space graph can be constructed from the set of operators and a given state by applying all possible operators to the given state, repeating the process for all newly created states. For a planning problem to be solvable it is necessary that one of the goal states can be constructed in this manner from the initial state.

Generally, when we speak of abstraction in planning, it is implied that one or more elements of a planning problem is being abstracted, i.e. either the initial state, one or more of the operators, or the goal states. As we shall see, it has generally been the case that planning systems have concerned themselves with abstracting operators.

Abstraction has been used as a mechanism in planning for two reasons. First, it is a natural extension, and one which people often use when doing everyday planning. Second, it is likely that planning with abstractions improves the efficiency of the planner.

1.2 Previous Work

1.2.1 STRIPS Assumption

Early work in planning [Green, 69] led to the discovery of severe deficiencies in trying to apply theorem proving to the planning problem. One of the main problems was the need for frame axioms [McCarthy and Hayes, 1969], axioms which stated what remained true from one state to the next. As it turns out many such axioms are generally needed for most planning problems.

STRIPS [Fikes and Nilsson, 1971] embodied one approach for dealing with this problem. In STRIPS, operators were structures consisting of a precondition list, a delete list, and an add list. States were sets of well-formed formulas. An operator was applicable in a state if all the items in the precondition list could be unified with members of the state. The result of applying an operator was that items in the delete list were deleted from the state, and items in the add list were added to the state.

These operators embodied what has come to be known as the STRIPS assumption - that whatever is not explicitly listed as an effect in an operator is automatically copied to the new state. The STRIPS assumption has proven an effective approach to the frame problem, obviating the need for time consuming frame axioms. Virtually all subsequent planners make use of the STRIPS assumption in some form. Some of the newer planners relax the STRIPS assumption somewhat in return for more flexibility in operator representation [Wilkins, 1988].

When subsequently we refer to operators we will mean STRIPS style operators, unless we specify otherwise. We now discuss some of the previous work on abstracting planning problems.

1.2.2 Macro Operators

One way an implicit state space graph can be abstracted is by defining new operators. One useful mechanism is to create new *macro operators*, each of which is the result of composing a sequence of two or more operators. Doing so allows the planner to traverse the state space graph more quickly, since intermediate, and presumably unimportant states, can be bypassed when transforming one state into another state.

The use of macro operators can be found in one of the earliest discussions on the use of abstraction in problem solving, namely Amarel's classic paper [Amarel, 1968]. Amarel traces a solution to the Missionaries and Cannibals problem which involves the introduction of macro operators, as well as other forms of abstraction which we will discuss later. Through these methods Amarel demonstrates how a seemingly complicated problem can be transformed into a trivial one by exploiting useful properties of the state space graph.

One of the first examples of an abstraction method being implemented in a planner is the use of MACROPS in STRIPS [Fikes et al, 1972]. A MACROP is a macro operator, composed automatically by STRIPS from a successful plan. This is done by storing the plan in a *triangle table* and then generalizing it by turning constants into variables. This generalized plan becomes a MACROP and can then be used by STRIPS to speed up planning considerably.

1.2.3 Hierarchical Operators

Although macro operators proved successful in STRIPS, their sequential was a limitation. NOAH [Sacerdoti, 1977] introduced the idea of least commitment to planning. Plans in NOAH were no longer represented simply as linear sequences of operators, but rather as partial orders of operators, compactly representing a set of total orders. These partial orders were represented in a *procedural net*. NOAH is the first example of a planner that searches in a space of partially ordered plans rather than the base state space provided by the domain. Strictly speaking, NOAH did not *search* this space, since no backtracking mechanism was incorporated.

In NOAH, macro operators were generalized, so that they no longer were simply compositions of sequences of operators, but rather were procedural encodings which, when executed, would produce a portion of a procedural net. These operators are termed *hierarchical* operators, and executing their procedural encodings is referred to as *expanding* an operator. The actual result of an operator expansion in NOAH might depend on the situation in which the operator is expanded, so there is not a simple one to one relationship between the hierarchical operators and partial orders of base level operators.

The use of hierarchical operators has been by far the most commonly used abstraction mechanism in planning since the advent of NOAH. Almost all ensuing planners employ some form of hierarchical operators, including NONLIN [Tate, 1977] and SIPE [Wilkins, 1988].

Unlike MACROPS in STRIPS, there is no example of a planner which learns hierarchical operators from the basic domain operators. Rather, they must be encoded by the user of the planner.

Because macro operators and hierarchical operators are defined in terms of other operators, these mechanisms will be termed *operator abstraction* mechanisms.

1.2.4 State Abstraction

Another planner that utilizes abstraction is ABSTRIPS [Sacerdoti, 1974]. It is based on STRIPS [Fikes and Nilsson, 1971] and utilizes a different abstraction mechanism from operator abstraction. Before proceeding it is important to have an understanding of the basic planning mechanism employed by ABSTRIPS.

ABSTRIPS abstracts by assigning criticalities to predicates. A criticality indicates the relative difficulty of making the particular predicate true in the domain, with the highest criticalities being assigned to predicates which cannot be affected by the planner.

Planning in ABSTRIPS proceeds in a "length-first" manner. At each stage a threshold criticality is determined. The planner then performs a complete STRIPS planning procedure, the sole difference being that predicates which have criticalities less than the current threshold are assumed to hold. The idea behind this is that those predicates can in some sense be considered details and can be achieved relatively easily. After each pass ABSTRIPS lowers the criticality threshold and proceeds to refine the current plan by attempting to achieve predicates which are now above the threshold. Planning is complete once the threshold is less than the smallest criticality. Using this method, Sacerdoti was able to achieve great speedup on some problems when compared to the bare-bones STRIPS system. The criticality values were generated in a semi-automatic manner, with the user providing a partial order of predicates which ABSTRIPS would use in assigning criticality values.

Since at any planning level, the preconditions of operators that have a criticality lower than the threshold will be dropped, each planning level consists of a new, abstract set of operators. Because an operator at one level is applicable in a superset of states from its corresponding operators at lower levels, this form of abstraction is termed *state abstraction.*

Tenenberg [Tenenberg, 1988] extends the ABSTRIPS representation by including criticality levels in the add and delete lists of operators. This, combined with a restricted method for computing the criticalities of predicates, produces a system which guarantees that planning at all levels of abstraction is consistent, which was not the case in the original ABSTRIPS system.

The problem is that if one is not careful when assigning criticality values it is possible to generate plans at higher levels of abstraction which result in inconsistent states. For example, Tenenberg presents the problem depicted in figure 1.2 in his thesis.

In the example the set L defines the language of the system, the set E is the set of essential predicates, namely those predicates which can be manipulated by the operators, O is the set of operators, K is a set of domain axioms, and crit is the set of criticality mappings on the predicates.

If the initial situation is described by

$$\{On(A,B), Clear(A), Holding(C)\}$$

L: (constants = {A, B, C}), (variables = {x, y, z}) (functions = ∅),
(predicates = {On, Clear, HandEmpty, Holding, *})

E = {On, Clear, HandEmpty, Holding},

O = {unstack(x, y)
P:{On(x, y), Clear(x), HandEmpty},
D:{On(x, y), Clear(x), HandEmpty},
A:{Holding(x), Clear(y)},

stack(x, y)
P:{Clear(y), Holding(x)},
D:{Clear(y), Holding(x)},
A:{On(x, y), Clear(x), HandEmpty}}

K = { Holding(x) ∧ y * x ⊃ ¬Holding(y),
HandEmpty ⊃ ¬Holding(x),
Clear(x) ⊃ ¬On(y, x),
On(x, y) ⊃ ¬On(y, x),
¬On(x, x),
A * B, B * C, A * C}.

crit = {<On, 1>, <Clear, 1>, <HandEmpty, 0>, <Holding, 0>}

Figure 1.2: ABSTRIPS Example from Tenenberg

then the plan $< unstack(A, B) >$ is applicable since the preconditions with criticality level 1 are satisfied. However, applying the plan results in the state

$$\{Holding(C), Holding(A), Clear(B)\}$$

which is inconsistent with the domain axiom which states that two blocks cannot be held at the same time.

Tenenberg proposes a way of assigning criticality values which guarantees that such inconsistencies do not occur. As we shall see later, our technique of predicate relaxation avoids this problem as well, albeit in a different manner.

Knoblock [Knoblock, 1990] uses a graph theoretic technique to identify dependencies among predicates in order to remove progressively more predicates at higher levels of abstraction. This results in abstractions which guarantee that if there is a plan at a high level of abstraction, there will be one at a lower level as well.

1.2.5 Discussion

In actuality, there is not such a clear dichotomy between the state and operator abstraction in their implementations. Hierarchical operators can be used to achieve state abstraction. This can be done by defining hierarchical operators which do not reference all the necessary predicates referenced by the operators, or their refinements, in their bodies. This is similar to the ABSTRIPS approach, but differs in that there is no enforcement of explicit abstract levels, as is the case with the explicit assignments of criticalities. Although more flexible, this lack of explicit criticality levels gives rise to "hierarchical promiscuity" [Wilkins, 1988], which can result in unnecessary planning. The problem arises when the planner tries to determine the truth value of a predicate, call it P. This is generally done by backchaining through the plan looking for places where P is changed. If operators are represented at different levels of abstraction there might be cases where the refinement of an operator results in a change to P, but this is not apparent at the current level of abstraction. In such cases, the truth value of P cannot be correctly determined. As we shall see later, there are several possible solutions to hierarchical promiscuity, but it remains a serious issue in planning with abstraction.

Furthermore, state abstraction generally results in the generation of new operators, which are used to generate the abstract state space. Although the exact differences between state and operator abstraction are not always obvious it remains a useful concept for distinguishing the two types of abstraction.

1.3 Theoretical Results on Abstraction

One might ask by how much abstraction improves planning efficiency. Empirically, it seems abstraction can be of great help. ABSTRIPS was able to achieve significant speedups in planning time as compared to STRIPS. The adherence to hierarchical operators in post-NOAH planners indicates that they are of great value in improving planning efficiency. There are also a few theoretical results to back up the value of abstraction in planning.

Korf [Korf, 1987] proves that under certain restrictive assumptions the optimum abstraction hierarchy of a state-space of size n consists of ln n levels of abstraction. Further, such an abstraction can reduce the expected search time from O(n) to O(log n). If n grows exponentially with the length of the plan, as is often the case, this result implies that the state space can be searched in linear time, given the abstraction hierarchy.

However, Korf makes one especially restrictive assumption, namely that a plan at one level maps directly to a plan at a lower level. This is usually not the case in planning, e.g. in ABSTRIPS. Rather, once a plan has been found at a higher level, more planning is necessary to develop the plan at the lower level. The plan at the higher level acts as a guide and proposes subproblems to be solved.

Knoblock [Knoblock, 1990] has analyzed this more general problem. However, it is difficult to arrive at useful results unless several new assumptions are made. The assumptions are as follows. First, if there is a solution at an abstract level, there is one at a lower level. This property is referred to as *downwards-compatibility.* Second, at one level of abstraction, each subproblem defined by the abstract level is independent, so that no backtracking is necessary. Third, every subproblem of an abstraction level is of the same size. Finally, the abstract planner produces the shortest solution.

Given these assumptions Knoblock derives that the worst case complexity of planning is reduced from $O(b^l)$ to $O(l)$, where b is the branching factor of the search space, and l is the length of the solution plan. This is analogous to Korf's result.

Korf's and Knoblock's results are very encouraging in that they suggest abstraction can transform intractable combinatorial problems into tractable ones. However, in practice, the restrictive assumptions made may not hold, and the results may not always be as spectacular. Specifically, the abstraction spaces may not satisfy downward-compatibility, meaning that a plan at a higher level has no expansion at a lower level. It is also possible that the subproblems defined at lower levels are not really independent, thus necessitating backtracking across subproblems at one level of abstraction. Nonetheless, these results provide an impetus for continued research on abstraction. It is also clear that human problem solving often makes use of abstractions when planning. Capturing this ability remains a strong motivation for pursuing research on abstraction.

1.4 Conclusion

Previous work on abstraction in planning falls under one of two rubrics, either operator abstraction, the most commonly used, or state abstraction. Hierarchical operators are generally considered useful for performing operator abstraction, although they can be used for state abstraction as well. ABSTRIPS was able to successfully perform state abstraction by assigning criticality values to predicates in the preconditions of operators. Finally, theoretical results suggest that planning can, in the best case, reduce planning time from exponential in the length of the plan down to linear.

Chapter 2

Predicate Relaxation

2.1 Introduction

In this chapter, we introduce *predicate relaxation*, a method for automatically performing state abstraction. The motivation for defining predicate relaxation is the need for determining whether a predicate should be considered a detail in a particular situation. As we have seen, ABSTRIPS accomplishes this by associating criticality values with predicates. However, whenever a predicate has a criticality value lower than the current planning threshold that predicate is true in all states of the search space. We will argue that whether a predicate should be considered a detail depends on the situation in which we are evaluating the predicate.

Consider the following example. Suppose we are planning a bus trip from one location in a city to another. When planning at a high level of abstraction, we would generally ignore the issue of whether or not we have adequate bus fares and concentrate on the route planning aspects of the problem. At high levels of abstraction we would like to consider having a bus fare a detail. However, if our plan is to be executed fairly soon, and we do not have exact change in our pockets, getting the exact bus fare might not be trivial, and we should not treat having the bus fare as a detail. However, if the first bus stop is close to a token booth, it should be relatively easy to obtain the bus fare, in which case it becomes a detail again. This, of course, presupposes that we have enough money to buy a token. If not, having bus fare ceases to be a detail again.

The point should be obvious. Whether or not having the bus fare should be considered a detail depends on the situation. In some situations obtaining the bus fare is trivial, in others it is more involved and requires some planning. ABSTRIPS's approach of using criticality values is unable to capture this context-dependency, and we are led to defining predicate relaxation.

The basic idea behind predicate relaxation is that, given a base level predicate P, a new predicate P^1_{rel} is defined which is true in a superset of states in which P holds. We say that P^1_{rel} is a *relaxed* version of P.

The above process can be repeated by defining P^2_{rel} from P^1_{rel}, and so on, creating a hierarchy of predicates. Of course, once a predicate has been relaxed to the point that it holds in all states there is no need to relax it further.

When we plan, if instead of using the original predicates, we use the newly defined relaxed predicates, we can decide if a predicate should be considered a detail by checking its relaxed definition. If the relaxation holds we say that the predicate is a detail and we do not plan for it. We will see later how this satisfies our requirement of context-dependency.

2.2 Computing Predicate Relaxation

Predicate relaxation defines a new predicate P^1_{rel} from a predicate P in such a way that P^1_{rel} holds in all states in which P holds and in all states in which P can be achieved by the application of one operator.

In order to precisely define predicate relaxation, regression must be introduced. Waldinger [Waldinger, 1977] introduced the technique of regression in the AI literature, although he credits [Manna, 1968, Hoare, 1969, King, 1969] with the original discovery.

Definition 1 *The regression $Reg(o,p)$ of predicate P over action o is the weakest relation that ensures the subsequent truth of P after executing o.*

Regression can now be used to define predicate relaxation.

In a domain with m operators, given a predicate P, we define P^n_{rel} as follows:

$$\begin{aligned} P^0_{rel} &= P \\ P^n_{rel} &= P^{n-1}_{rel} \bigvee_{i}^{m} Reg(Op_i, P^{n-1}_{rel}) \end{aligned}$$

where $Reg(Op_i, P)$ is the regression of predicate P through operator Op_i.

In general, P will have a non-zero arity, e.g. $Clear(x)$ or $On(A, B)$. When computing predicate relaxation, we will usually only do so once for each predicate, and replace its arguments with schema variables. For example, after computing the relaxation of $On(x,y)$, the result can be instantiated to the predicates $On(A, B)$, $On(C, D)$, etc. There is no need to compute the relaxation expression separately for each different predicate.

It is also the case that P^n_{rel} becomes more complex as n grows. It should be noted that the regression of P^n_{rel} is always computable, although the complexity of the computation may increase with the complexity of P^n_{rel}.

To improve the efficiency of computation one can check before regressing a predicate P that it appears in the add list of the operator. If it does not the regression is not necessary (the resulting expression would simply be subsumed by P). This does not mean that we never regress predicates through operators where they do not appear in the add list. If we are regressing $P \wedge Q$ through an operator where P appears in the add list, we must also regress Q through the same operator, even though it might not appear in the add list.

In many cases the regression of a predicate through an operator will be equivalent to the preconditions of the operator with the appropriate variable instantiations. However, there are possible complications. For example, suppose we are regressing the predicate $P(x)$ through and operator and $P(y)$ appears in the delete list of the operator, but not in the add list. Then, one of the conjuncts in the resulting expression will be $x \neq y$, to guarantee that $P(x)$ is not deleted by the application of the operator.

We will see later how we can use relaxed predicates to considerably speed up planning. We now provide an example to help the reader become familiar with predicate relaxation.

2.3 Predicate Relaxation Example

Suppose we have a blocks-world system with the following four operators, where P is the precondition list, D is the delete list, and A is the add list.

Pickup(x)
    P:{Clear(x),Handempty}
    D:{Clear(x),Handempty}
    A:{Holding(x)}

Putdown(x)
    P:{Holding(x)}
    D:{Holding(x)}
    A:{Clear(x),Handempty}

Stack(x,y)
    P:{Clear(y),Holding(x)}
    D:{Clear(y),Holding(x)}
    A:{On(x,y),Clear(x),Handempty}

Unstack(x,y)
    P:{On(x,y),Clear(x),Handempty}
    D:{On(x,y),Clear(x),Handempty}
    A:{Holding(x),Clear(y)}

To simplify the example we assume only two blocks A and B.
The predicates would be relaxed as follows:

$$Handempty^1_{rel} = Handempty \vee Holding(x) \vee (Holding(y) \wedge Clear(z))$$

which can be simplified to:

$$Handempty^1_{rel} \doteq Handempty \vee Holding(x)$$

which in turn can be reduced to:

$$Handempty^1_{rel} = T$$

assuming normal domain constraints.
Here we see that *Handempty* can easily be guaranteed to hold.
Proceeding,

$$Clear^1_{rel}(x) = Clear(x) \vee Holding(x) \vee (Clear(y) \wedge Holding(x))$$

$$\vee (On(z, x) \wedge Clear(z) \wedge Handempty)\ .$$

$$Clear^1_{rel}(x) = Clear(x) \vee Holding(x) \vee (On(z, x) \wedge Clear(z) \wedge Handempty)$$

Although it might not be obvious at first glance, using appropriate domain constraints and the fact that we have only two blocks the above formula reduces to:

$$Clear^1_{rel}(x) = T$$

Next we relax $Holding(x)$:

$$Holding^1_{rel}(x) = Holding(x) \vee (Clear(x) \wedge Handempty)$$

$$\vee (On(x, y) \wedge Clear(x) \wedge Handempty)$$

$$Holding^1_{rel}(x) \equiv Holding(x) \vee (Clear(x) \wedge Handempty)$$

At the next step we relax $Clear(x) \wedge Handempty$:

$$Holding^2_{rel}(x) = Holding(x) \vee$$

$$((Clear(x) \vee Holding(x)) \wedge (Handempty \vee Holding(z)))$$

Note that unlike our independent derivation of $Clear^1_{rel}$ we could not use the preconditions of the Unstack operator, since Unstack clobbers *Handempty*.

We can simplify:

$$Holding^2_{rel}(x) = Holding(x) \vee Clear(x)$$

$$Holding^3_{rel}(x) = Holding(x) \vee Clear(x) \vee (On(z,x) \wedge Clear(z) \wedge Handempty)$$

Using the fact that we have only two blocks this reduces to:

$$Holding^3_{rel}(x) = T$$

All that remains is to relax $On(x,y)$.

$$On^1_{rel}(x,y) = On(x,y) \vee (Clear(y) \wedge Holding(x))$$

As before we can replace $Clear(y) \wedge Holding(x)$ by $Holding(x)$.

$$On^1_{rel}(x,y) = On(x,y) \vee Holding(x)$$

Then, relaxing $Holding(x)$,

$$On^2_{rel}(x,y) = On(x,y) \vee Holding(x) \vee (Clear(x) \wedge Handempty)$$

Relaxing $Clear(x) \wedge Handempty$,

$$On^3_{rel}(x,y) = On(x,y) \vee Holding(x) \vee (Clear(x) \wedge Handempty) \vee Holding(y)$$

Finally, relaxing $Holding(y)$ we get,

$$On^4_{rel}(x,y) = On(x,y) \vee Holding(x) \vee (Clear(x) \wedge Handempty)$$
$$\vee Holding(y) \vee (Clear(y) \wedge Handempty)$$

In our simple two blocks domain this reduces to:

$$On^4_{rel}(x,y) = T$$

2.4 Discussion of Predicate Relaxation

It should be obvious that if P^n_{rel} holds in a state, there is a plan which can achieve P in n steps or less. The plan is just the sequence of operators through which P was regressed to arrive at the expression that holds in the current state. However, because we likely simplified the regressed expression along the way, we do not necessarily know what this plan is. We just know that there is indeed such a plan. By the definition of regression, this expression being true guarantees that P will hold after the application of that sequence of operators.

If P^n_{rel} holds, but Q^n_{rel} does not, one can say that P^n_{rel} is more of a "detail" than Q^n_{rel}, since P can be achieved more easily than Q. Predicate relaxation provides a gradual widening of the states in which a predicate holds. In ABSTRIPS, a predicate can either hold in those states in which it was intended to hold, or, when its criticality value is less than the current threshold, hold in all states of the domain. This change in the semantics of a predicate can be quite sharp. Predicates abstracted with predicate relaxation, however, avoid this semantic cliff, since the set of states in which they hold is gradually enlarged at each relaxation level.

Unlike ABSTRIPS, the abstraction hierarchy is computed automatically. In ABSTRIPS the user had to supply a partial order of predicates which was used to compute criticality values. It is not always obvious what this partial order should be.

Also, besides the semantic cliff that ABSTRIP's method suffers from there is a more subtle problem. Because of the way criticality values are computed by ABSTRIPS it often happens that one predicate will have different criticality values in the preconditions of different operators. For example, in the example presented in [Sacerdoti, 1974] the following operators are presented:

Gothrudr(R,d,ry)

```
    P:{[6]Type(d,Door),[5]Inroom(R,rx),[6]Connects(d,rx,ry),
        [2]Status(d,Open),[6]Type(ry,Room)}
    D:{Nextto(R,$1),Inroom(R,rx)}
    A:{Inroom(R,ry)}
```

Close(R,d)

P:{[6]Type(d,Door),[5]Nextto(R,d),[5]Status(d,Open)}
D:{Status(d,Open)}
A:{Status(d,Closed)}

In the Gothrudr operator the Status(d,Open) precondition has a criticality value of 2, whereas in the Close(R,d) it has a criticality of 5. This means that in the same situation, when planning at a criticality threshold between 2 and 5, ABSTRIPS treats Status(d,Open) as a detail for one operator, but as an important predicate that needs to be planned for in another operator. This type of inconsistency does not happen with predicate relaxation.

In the next chapter we will have more to say about the differences between planning with predicate relaxation and ABSTRIPS.

2.4.1 Context-dependency

It should be clear that predicate relaxation can be used to satisfy our requirement that the *detailness* of a predicate should depend on the situation in which the predicate is being evaluated. Using our previous example of the bus fare, at abstraction level n, we will consider $Have(BusFare)$ to be a detail in any situations in which $Have^{n}_{rel}(BusFare)$ holds. In this manner, $Have(BusFare)$ will be considered a detail only in those situations in which there is a plan of length n or less to achieve $Have(BusFare)$. This seems to be a reasonable criterion for determining when a predicate should be considered a detail.

2.5 Summary

We have introduced *predicate relaxation*, a method for defining hierarchies of predicates. Predicate relaxation is a technique for performing state abstraction. We have also compared predicate relaxation to ABSTRIPS's technique of computing criticality values. The basic motivation for using relaxed predicates is the context-dependency of *detailness*. Using predicate relaxation gives us a means for determining in which situations a predicate should be considered a detail.

In the next chapter we will see how the hierarchies generated by predicate relaxation can be used to significantly improve planning efficiency.

Chapter 3

PABLO

3.1 Introduction

Since the advent of NOAH [Sacerdoti, 1977] much of planning research has concerned itself with developing representationally powerful planners. Researchers have produced planners that are quite encompassing in the domains they can represent, incorporating resource-based reasoning, temporal reasoning, and other techniques to facilitate the encoding of domains.

NOAH has had a great influence on modern day planning research. Virtually all subsequent planners employ some of the techniques introduced in NOAH, the most distinguishing one being the encoding of plans in *procedural nets.* Procedural nets provide a convenient representation for plans. They allow the planner to represent plans as partial orders, rather than as linear sequences as had previously been the case, which allows NOAH to postpone commitment to any particular action ordering until absolutely necessary.

One important characteristic of the procedural net is that it encodes procedural as well as declarative information. The procedural data is stored in terms of user defined functions (SOUP code functions, in the case of NOAH) for expanding nodes in the procedural network at the next planning level.

A *procedural net* is procedural precisely because it not only encodes information about the problem at hand, but also because it encodes information on *how* the problem is to be solved.

However, with this focus on representational power, there has also been a shift away from general, domain independent problem-solving. NOAH signaled this shift by providing no backtracking search capability. Thus, if NOAH mistakenly chose a wrong operator with which to expand a subgoal, it had no provision for backing up and attempting another operator.

Because of its lack of backtracking, a plan in NOAH is basically unfolded from the operator definitions. It is the responsibility of the user to provide NOAH with correct and detailed enough SOUP functions so that the planning problem can be solved without backtracking. NOAH can be viewed as a *programming language* for writing programs that compute plans composed of primitive actions.

It is important to note that NOAH's lack of a backtracking mechanism, which at first glance appears to be a serious omission, is closely tied to the planning philosophy embodied in NOAH. Of course, the least-commitment principle embodied in the procedural net, allowed NOAH to avoid many dead-ends that purely linear planners would have encountered, thus further reducing the need for a search capability.

However, just as importantly, unlike previous planners, the aim was to provide a framework wherein the user could apply domain-specific knowledge to solve complex planning problems.

NOAH shifted a major part of the problem-solving responsibility from the planner, where it had previously resided, squarely onto the shoulders of the user. This had the advantage of greatly enhancing the computational efficiency of NOAH as compared to previous planners.

Of course, not all the problem-solving responsibility lies with the user. There is, after all, much declarative information in the procedural net that NOAH makes use of. Specifically, after each level is expanded a set of critics examines the current state of the plan and modifies it in case of difficulties, e.g. the possible clobbering of a precondition by an action.

It has generally been assumed that this division of labour between the domain-specific SOUP functions and the domain-independent critics provided an adequate compromise between the conflicting requirements of completeness and efficiency in planning. We will argue that this position needs to be re-examined. We believe there is still much to be done in the area of developing powerful planners and that it might be necessary to eventually endow planners with more powerful domain independent techniques.

This is based on the belief that planners should strive to provide as much problem-solving aid as possible to the user attempting to solve a planning problem. The more burden we place on the encoder of the domain, the less valuable a tool the planner becomes. Ideally, when faced with a new domain, the user should not have to discover the efficient algorithms for solving problems in that domain, but should be able to simply provide the planner with a naive encoding of the domain objects and primitive actions.

For example, if faced with the Towers of Hanoi problem for the first time it does not seem reasonable to expect the encoder of the domain to know about efficient algorithms for solving the problem. If she did indeed know such algorithms it would probably be more reasonable to encode them directly in a general programming language.

Rather, we can expect the encoder of the domain to provide descriptions of the objects and relations of the domain, e.g. pegs, disks, clear(disk), on(disk1,disk2), smaller(disk1,disk2), etc... The only action the encoder is likely to be aware of is the Move(disk1,peg1,peg2) action, namely move disk1 to peg1 from peg2. This level of information is realistically all that can be expected from the encoder of the domain. It is then up to the planner to make use of this domain description to facilitate the development of plans for solving problems in this domain.

Clearly, because of its lack of backtracking, NOAH is likely to fail to produce plans in this domain. Given only the naive encoding of the domain, search is inherently necessary in arriving at a solution. Of course, search is not the whole answer. If the planner merely provides a blind search capability, e.g. complete breadth-first search, it is not aiding the user of the system appreciably.

Therefore, it is not simply enough that the planner take responsibility for the problem-solving in a domain, it must do so in a non-trivial way to be of aid to the user.

Many of the advancements in planning since NOAH have been in the area of allowing more representational power by the encoder of the domain, but very few have been in the area of improving the basic problem-solving capabilities of planners. Blind search still seems to be the default for most planners.

3.1.1 Post-NOAH Planners

The next major planner after NOAH was NONLIN [Tate, 1977] developed by Austin Tate at Edinburgh. NONLIN improved on NOAH in several ways. Unlike NOAH, it searched the space of partial plans. It also provided a more perspicuous language in which to represent operators, as well as *typed preconditions*. However, its search is blind, making its usefulness somewhat questionable. Tate states [Tate, 1977]:

> We expect that the first choice taken should lead to a solution...if failure occurs with the first plan being considered, our experience is that backtracking can lead to long searches.

Unless the problem domain was encoded in such a way that the solution could be directly unfolded from the operator definitions, there was a slim hope of finding a solution in a reasonable amount of time.

SIPE [Wilkins, 1984] represents the state of the art in classical planning. In addition to the planning features discussed to this point, SIPE extends the plan representation in several ways. It allows for a deductive causal theory which greatly reduces the complexity of the operator descriptions. It provides capabilities for reasoning with resources, including time. In addition to this it provides a powerful constraint language which allows it to partially specify objects.

SIPE achieves a high level of efficiency and is the first planner to successfully be applied to real-world applications [Wilkins, 1988]. However, even SIPE could benefit from advancements in domain-independent problem-solving techniques to improve its search capability, since, as in previous planners, it remains blind, and is guided to a large extent by the user defined operators.

TWEAK, developed by Chapman [Chapman, 1987], is a formalization of earlier non-linear planners. Chapman introduces a sound and complete Modal Truth Criterion for determining the truth of predicates at any point in a non-linear plan. TWEAK is guaranteed to find a solution to a planning problem if one exists.

The above is by no means an exhaustive review of earlier work on planning; just a selection of some major systems, chosen to contrast the traditional planning research with our research on PABLO. For a good overview of planning systems see [Georgeff, 1987, Drummond and Tate, 1989, Allen et al, 1990].

3.2 Planning Terminology

To facilitate the description of PABLO we will use the TWEAK terminology. In this section we present some important definitions. A more extensive description can be found in [Chapman, 1987].

A planner is said to be *sound* if whenever it finds a solution to a planning problem, the solution plan is a correct plan for solving the problem. A planner is said to be *complete* if whenever there is a solution to a planning problem the planner can find it.

At the core of any nonlinear planner is the algorithm for determining the truth of predicates at a particular point in the plan. The condition under which a predicate is said to hold is known as a *truth criterion.* TWEAK introduced the first such criterion, namely the Modal Truth Criterion. In chapter 9 we introduce a new truth criterion.

Two variables are said to *codesignate* if they are constrained to always refer to the same domain object. Similarly, two predicates are said to codesignate if they are of the same type and their respective arguments codesignate. For example, $On(x, y)$ and $On(v, w)$ codesignate if x and v as well as y and w codesignate.

Each action in a plan is an instantiated operator. Each action defines two *situations*, namely the situation immediately preceding the action and the situation immedicately following the action. A *plan* is a partial order of actions with an initial situation and a final situation.

A predicate is said to be *asserted* in a situation if it codesignates with a member of the add list of the action immediately preceding the situation. A predicate is asserted in the initial situation if it codesignates with a member of the initial situation. A predicate is *denied* in a situation if it codesignates with a member of the delete list of the action immediately preceding the situation.

A *goal* is a pair consisting of a predicate and a situation in which that predicate must hold. An action is said to *establish* a goal if the predicate of the goal is asserted in the situation immediately following the action.

An action is said to *clobber* a goal if it occurs after the establisher of the goal and before the situation in which the goal predicate must be true and it denies the goal predicate.

An action is said to be a *white knight* if it occurs after a clobberer and before the situation in which the goal predicate must be true, and whenever the clobberer clobbers the goal predicate the white knight establishes it..

This brings us to the notions of necessity and possibility. A plan can generally be completed in many ways, depending on which temporal and codesignation constraints are added to it. If a property of the plan holds in all completions we say it *necessarily* holds. If it holds in some completions we say it *possibly* holds. For example, if an action clobbers a goal in all completions of a plan we say that the action necessarily clobbers the goal in the plan. If it only clobbers the goal in some completions of the plan, we say the action possibly clobbers the goal.

3.3 Overview of PABLO

We can provide more problem-solving capability in a domain-independent planner and thereby shift the burden of problem solving from the user to the planner, by analyzing the encoding of the domain before beginning the actual planning process.

At one extreme, the planner could simply generate the whole search space ahead of time, thus trivializing the planning process. This is essentially the approach taken in *Universal Planning* [Schoppers, 1987]. There are several problems in attempting this, which we shall return to later in this thesis.

Another approach is to employ predicate relaxation. By relaxing predicates in the domain, we discover relevant facts about each predicate's difficulty. This is the approach taken by PABLO.

3.4 Underlying Planning Algorithm

PABLO uses iterative-deepening search [Korf, 1985] coupled with TWEAK's Modal Truth Criterion as its underlying planning algorithm. We chose this algorithm primarily because of its provable correctness and completeness. A breadth first implementation requires too much space so an iterative deepening approach was adopted. See table 3.1 for a high level description of the algorithm.

One thing to note about the above algorithm is that for every call to *plan* we only consider resolving one outstanding goal, even though there might be several which are unachieved. The reason we can do this, is that if we fail in solving for one goal, trying to solve for any of the other outstanding goals first can have no synergistic effect in solving the original goal. This is because the order in which goals are attempted is irrelevant, as all possible establishers are available to the algorithm at any one point in the form of operator templates which we can instantiate into the weakest form of an action. Adding constraints to an action can never result in the possible establishment of a proposition that could not already be possibly established by the action as it was first instantiated. Therefore, if we fail to solve an unachieved goal, we might as well backtrack, since continued work on other goals will not result in the possible achievement of the original goal.

For reasons of simplicity we have omitted declobbering by white knight in the overall control structure. See [Chapman, 1987] for an extensive discussion of the role of white knights in planning. This procedure is complicated and omitting it does not affect the completeness of the planner. The reason for this is that final plans always have all their variables codesignating with exactly one constant. Therefore, a white knight either asserts the proposition, in which case we would use it as an establisher, or it does not, in which case we can use separation to declobber the goal.

```
CallPlan ()
        maxdepth ← 1
        loop forever
            plan(0 maxdepth)
            maxdepth ← maxdepth + 1


Plan (opcount maxdepth)
    g ← any unachieved goal in the plan
    if g then
        for s one of all possible situations that can establish g
            constrain s to be before the situation g.p has to hold
            for p one of all the predicates asserted in s
                add codesignation constraint p ≈ g.p
                declobber(g,s,opcount,maxdepth)
                remove codesignation constraint p ≈ g.p
            remove constraint that s be before the situation g.p has to hold
        if (opcount < maxdepth) then
            for op one of the possible operators that can establish g
                instantiate and insert op into the plan
                constrain op to be before the situation g.p has to hold
                for p one of all the predicates asserted by op
                    add codesignation constraint p ≈ g.p
                    declobber(g,s,opcount+1,maxdepth)
                    remove codesignation constraint p ≈ g.p)
                remove op from plan
    else print(plan)
        break
```

Table 3.1: Base Level Control Structure of PABLO

```
Declobber (g,establisher,opcount,maxdepth)
        clobberer ← any step in the plan which clobbers g
        if clobberer then
                constrain clobberer to be before establisher
                declobber(g,establisher,opcount,maxdepth)
                remove constraint that clobberer be before establisher
                constrain clobberer to be after situation in which g has to hold
                declobber(g,establisher,opcount,maxdepth)
                remove constraint that clobberer be after
                situation in which g has to hold
                for p one of the possible predicates asserted by
                clobberer which can deny g.p
                    add codesignation constraint p ≉ g.p
                    declobber(g,establisher,opcount,maxdepth)
                    remove codesignation constraint p ≉ g.p
        else
                plan(opcount,maxdepth)
```

Table 3.2: Base Level Control Structure continued.

3.4.1 Plan Representation

PABLO uses a modified version of TWEAK's modal truth criterion during planning. Its plan representation is based on the TWEAK plan representation. In chapter 7 we will extend this representation to include hierarchical operators. In the name of efficiency some extensions have been provided to the operator representation. Specifically, preconditions can be specified so as not to be planned for by PABLO, but rather, just checked before the application of an operator. Further, propositions in the add list of an operator can be specified to be side effects of the operator and should not be considered as possible establishers for unachieved goals. Finally, variables of propositions in delete lists can be specified to be *global*, resulting in a simple form of universal quantification. Each of these extensions is completely optional, but can be used to significantly improve efficiency.

3.4.2 Relaxation Phase

Before the planning process begins, PABLO performs a relaxation phase, wherein it creates the relaxed definitions for the predicates appearing in the postconditions and preconditions of operators. This need only be done once for each domain.

The user of PABLO specifies the level to which the predicates should be relaxed. PABLO creates a relaxation definition for each different type of predicate in the domain. This definition is a relaxation *schema* and is instantiated every time a predicate is instantiated during planning.

For example, just as was done in the blocks world example of chapter 2, relaxation schemas would be created for the predicates Clear(x), Handempty, Holding(x) and On(x,y). Then, during planning, a particular predicate instance, say On(A,B) would have associated with it an instance of the relaxation schema for On(x,y) where x has been bound to A, and y has been bound to B.

As was done in the example presented during the description of predicate relaxation considerable simplification can be accomplished with the use of domain constraints. We will return to the issue of simplifying predicate relaxation expressions in chapter 8.

3.4.3 Planning Phase

The general idea during planning is that PABLO first consider the most important predicates, and then consider successively less important predicates. This is accomplished by associating each planning level with a relaxation level, and planning with relaxed predicates of that level. At any particular planning level, any predicate whose relaxation definition is true at that level is considered a detail and is not specifically planned for.

The operator Pickup(x), previously discussed, becomes at level 1,

Pickup(x)
  P:{$\text{Clear}^1_{\text{rel}}$(x),$\text{Handempty}^1_{\text{rel}}$}
  D:{Clear(x),Handempty}
  A:{Holding(x)}

When moving down abstraction levels, if newly created subgoals appear in different sections of the plan, PABLO attempts to achieve them independently. The rationale for this being that these predicates were considered "details" at the higher level and presumably do not have global consequences. In cases where this assumption fails, the consequences can, of course, be costly, in terms of computation time. However, in our experience the increased efficiency outweighs this risk.

The above is accomplished by beginning with the initial situation and any instantiated operators that do not have another instantiated operator necessarily between the initial situation and itself. PABLO plans for any outstanding preconditions or goals in this segment of the plan. Once this is done the plan is augmented with the next set of instantiated operators which do not have another operator which is necessarily before themselves. This process is repeated until the final situation is included in the plan. Once the full plan has been expanded we can move on to the next abstraction level.

3.4.4 Truth Criterion

At the core of any planning system is the procedure for determining the truth of predicates. Because PABLO has a restricted plan representation, the introduction of abstract predicates complicates this computation somewhat. If arbitrary first order sentences were allowed in the postcondition of operators, the definition for each abstract predicate could be included in the initial situation. This rule would then be propagated to every situation in the plan. No extra mechanism would then have to be provided in the planner to deal with abstract predicates.

Due to the restriction on PABLO's representational capability we extend the definition of asserts to include abstract predicates. In a base level system, a predicate is asserted in a particular situation if one of two conditions holds. If the situation is the initial situation then the predicate must be contained in the situation description. Otherwise, the predicate must be contained in the add list of the operator immediately preceding the situation. We extend these conditions as follows in order to accommodate for relaxed predicates.

Definition 2 *A relaxed predicate P^n_{rel} is asserted in situation s iff $\mathcal{T}(s) \vdash P^n_{rel}$, where $\mathcal{T}(s)$ is the theory consisting of the base level predicates which are necessarily true in s.*

The computation of the predicates that necessarily hold in s is then done with TWEAK's modal truth criterion. This criterion is somewhat conservative in determining the truth of abstract predicates but has proven quite adequate in practice. We shall later define a new truth criterion which is valid for a more expressive plan representation.

3.5 Examples of planning with PABLO

3.5.1 Towers of Hanoi

A well known problem with many inherent abstractions is the Towers of Hanoi problem. The operator given to PABLO is the following:-

Move(x,z)
    P:{Smaller(x,z),Movable(x),On(x,y),Clear(x),Clear(z)}
    D:{On(x,y),Clear(z)}

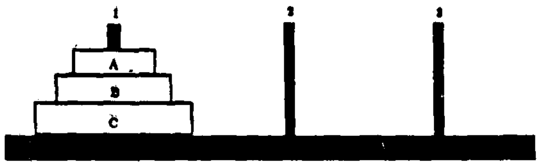


Figure 3.1: Towers of Hanoi

A:{On(x,z),Clear(y)}

See figure 3.2 for a trace of PABLO solving the three disk Towers of Hanoi problem. The plan at the highest level of abstraction consists of Move(C,P3). At this level all its preconditions are satisfied ($\text{Clear}^2_{\text{rel}}(C)$ is satisfied since it can be achieved in two steps).

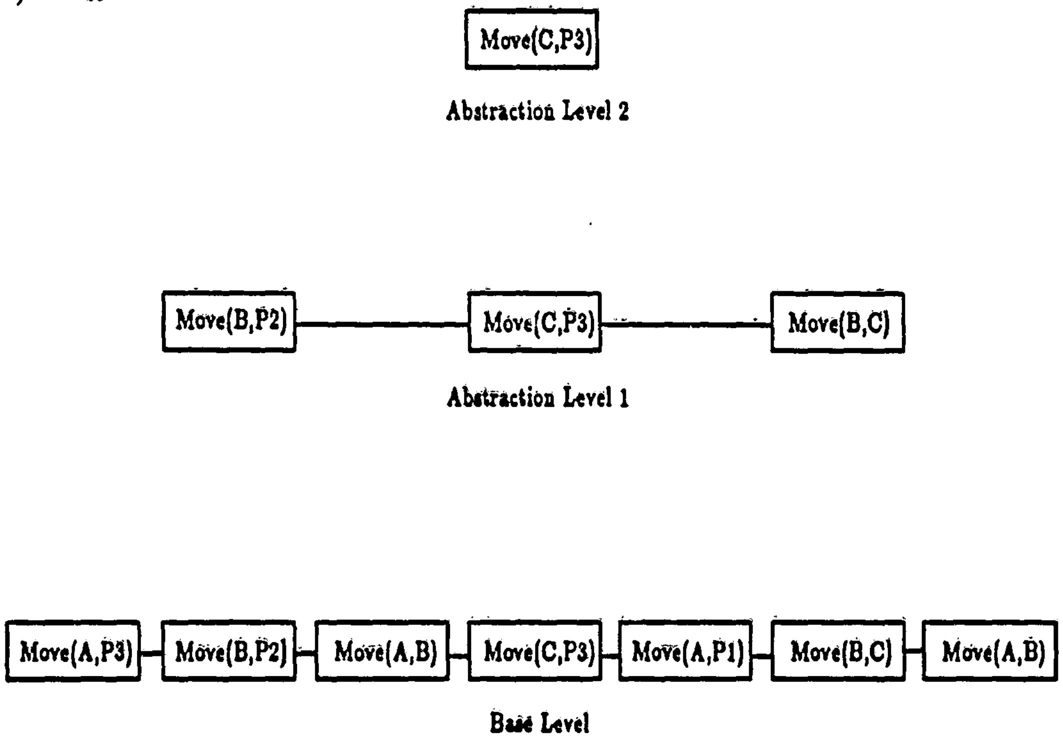


Figure 3.2: Trace of PABLO solving the Towers of Hanoi

When we move down to the next abstraction level $\text{Clear}^2_{\text{rel}}(C)$ becomes $\text{Clear}^1_{\text{rel}}(C)$ which is not satisfied in our initial state, since we cannot clear C in one step. PABLO therefore plans to achieve $\text{Clear}^1_{\text{rel}}(C)$ by adding the action Move(B,P2) to the plan. In doing so it undoes $\text{On}^1_{\text{rel}}(B, C)$, which PABLO then plans to reachieve, using the action Move(B,C). At this point the plan at the first level of abstraction is complete since all the first level relaxations of the goals and preconditions are satisfied. Planning is then completed at the base level using the original predicates of the domain.

In this case PABLO has discovered and made use of the inherent abstractions in the domain. Including the time it takes to generate the relaxation definitions, which in this example is negligible, PABLO solves the problem over 100 times faster using the abstractions than without using them. In chapter 5 we will present more comprehensive empirical results of PABLO.

3.5.2 Comparison with Other Planners

All planners of which we are aware, with the exception of ABSTRIPS, would have to revert to a full backward search of the state space if given this example.

Planners using operator abstraction can reason abstractly about this problem only if new operators are defined by the encoder of the domain, a task which might be both time-consuming and prone to errors. As we have pointed out earlier, in this research we are striving to provide powerful problem-solving capabilities given simple encodings of the domain.

ABSTRIPS assigns the following criticality values to the predicates in the domain:

```
Move(x,z)
    P:{{3}Smaller(x,z),{3}Movable(x),{2}On(x,y),{2}Clear(x),{2}Clear(z)}
    D:{On(x,y),Clear(z)}
    A:{On(x,z),Clear(y)}
```

ABSTRIPS creates only one level of abstraction in this domain. When solving the problem, after finishing the abstract level, the plan consists of one action Move(C,P3). Although this is of some aid in developing the plan at the base level it is not as useful as PABLO's hierarchy.

ABSTRIPS's hierarchies are domain-dependent but problem-independent. The number of different criticality values, and therefore the number of abstraction levels, of ABSTRIPS is constrained by the number of different predicates of the domain. For the 4 disk Towers of Hanoi, ABSTRIPS still has only one abstraction level, whereas PABLO generates 3 abstraction levels. In general, for the n-disk Towers of Hanoi problem, PABLO generates $n-1$ abstraction levels, whereas ABSTRIPS still creates only one abstraction level.

3.6 Blocks World

3.6.1 Example Problem

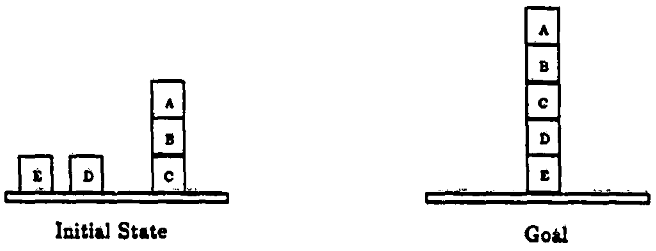


Figure 3.3: Blocks World

This version of the blocks world has two operators:

PUTON(x,y)	TABLEOPR(x,y)
P:{Clear(x),Clear(y),On(x,z)}	P:{Clear(x),On(x,y)}
D:{Clear(y),On(x,z)}	D:{On(x,y)}
A:{Clear(z),On(x,y)}	A:{Clear(y),On(x,TABLE)}

The goals are On(A,B), On(B,C), On(C,D), and On(D,E). PABLO begins planning at abstraction level 2. See figure 3.4 for a trace of PABLO solving this problem.

At abstraction level 2 the only goal not satisfied is $\mathrm{On}^2_{\mathrm{rel}}(\mathrm{C},\mathrm{D})$. PABLO plans to achieve this goal using the action Puton(C,D). All its preconditions are satisfied at this level of abstraction.

At abstraction level 1 the precondition $\mathrm{Clear}^1_{\mathrm{rel}}(\mathrm{C})$ is not satisfied so PABLO adds the action Tableopr(B,C) to achieve it. It then reachieves $\mathrm{On}^1_{\mathrm{rel}}(B,C)$ by adding the action Puton(B,C).

The plan is then completed at the base level using the base level predicates. Notice that the resulting plan is nonlinear. PABLO solved this problem 130 times faster with the abstractions than without them, including the time to generate the predicate relaxation definitions.

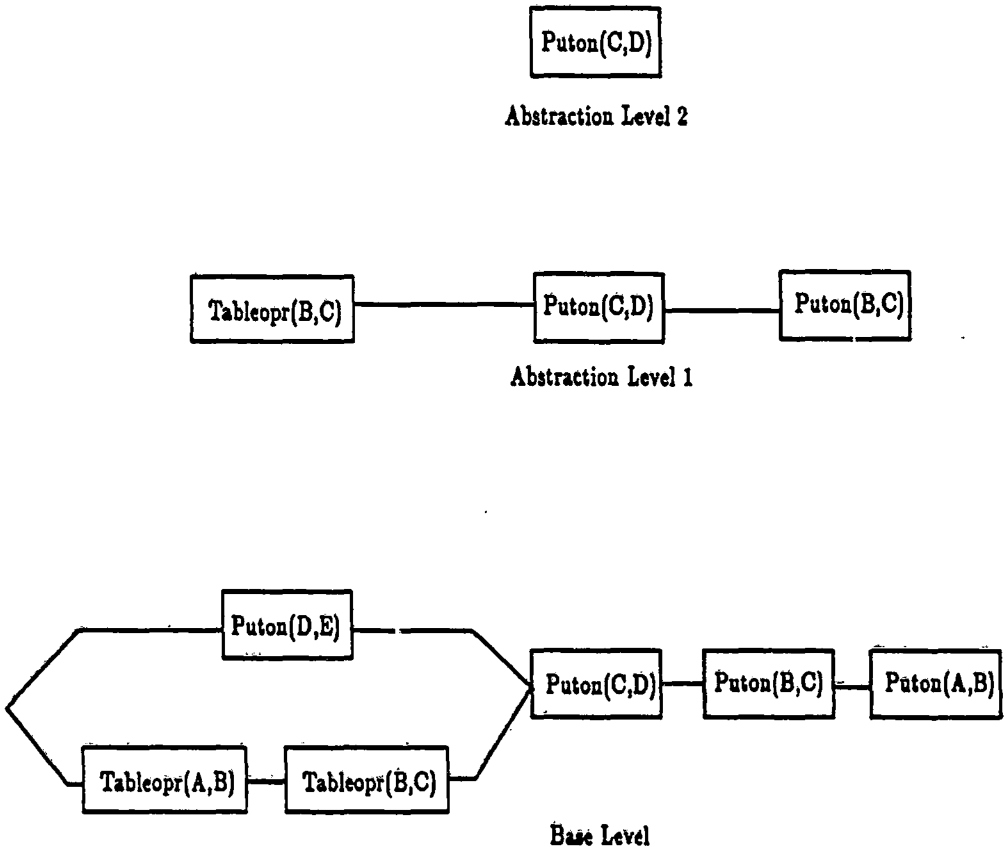


Figure 3.4: Blocks world trace.

3.6.2 Another Example

The following example shows the effect of nonlinear plans at higher levels of abstraction. See figure 3.5 for an illustration of the problem.

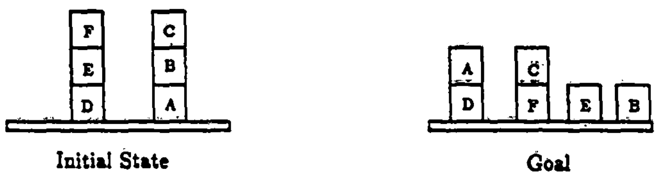


Figure 3.5: Blocks World Problem.

The goals of the problem are On(A,D) and On(C,F). A trace of PABLO solving this problem can be found in figure 3.6. At abstraction level 2 PABLO satisfies the goal $On^2_{rel}(A, D)$ by introducing the action Puton(A,D). Its preconditions and the goal $On^2_{rel}(C, F)$ are now satisfied at this level of abstraction.

At abstraction level 1 the preconditions $Clear^1_{rel}(A)$ and $Clear^1_{rel}(D)$ of the action Puton(A,D) are no longer satisfied. PABLO at this point inserts the actions Tableopr(x,A) and Tableopr(y,D). There is no reason to order these actions so they remain unordered. At this point the first half of the plan is completed. PABLO now considers the second half which includes the final situation. This has the effect of introducing the goal $On^1_{rel}(C, F)$ to this level. However, this goal is satisfied at this level of abstraction so no action is inserted to achieve it.

At the base level PABLO proceeds in three steps. The first step considers the preconditions to the two actions Tableopr(x,A) and Tableopr(y,D). PABLO first considers the precondition On(x,A), which is not satisfied at the base level since x is a variable. The variable x is then instantiated with On(B,A). At this point Clear(B) is considered since it is not satisfied at the base level. This is satisfied by inserting the operator Tableopr(C,B) to the plan. An analogous procedure ensues to satisfy the preconditions of Tableopr(y,D), resulting in the insertion of the operator Tableopr(F,E) and the instantiation of variable y with block E.

At this point this segment of the plan is fully satisfied and the preconditions to operator Puton(A,D) are now considered. However, these are all satisfied at the base level. Finally, the goals of the final situation are considered. The remaining goal On(C,F) is unsatisfied at the base level, so the operator Puton(C,F) is inserted. Its preconditions of Clear(C) and Clear(F) can be satisfied by constraining the operators

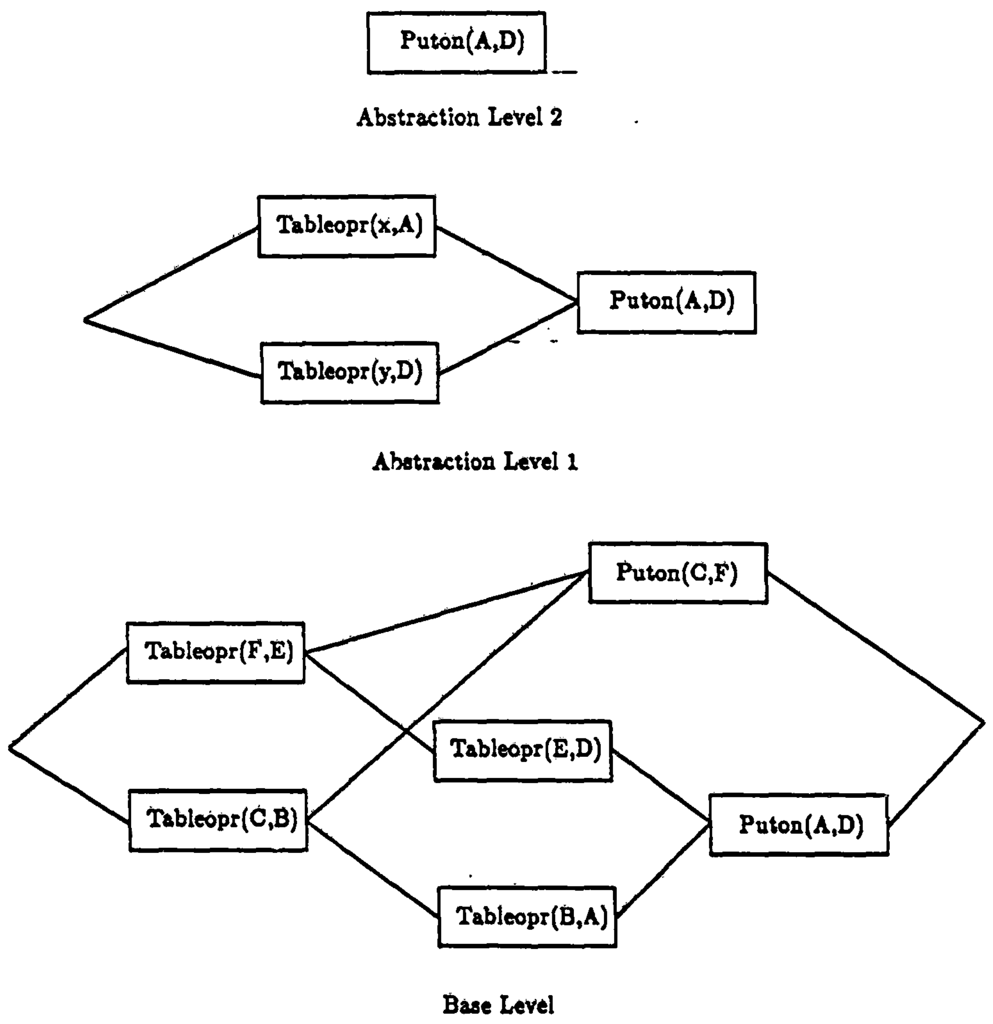


Figure 3.6: Nonlinear blocks world trace.

Tableopr(F,E) and Tableopr(C,B) to appear before Puton(C,F). At this point the plan is complete at the base level and PABLO terminates.

It should be noted that because of the partitioning of the plan at each level into self-contained subproblems that the possibility of suboptimal plans is introduced. In this example, PABLO produces a plan that is one step longer than optimal, since it could have satisfied the goal On(C,F) by the operator Puton(C,F) and at the same time satisfied the precondition Clear(B), thus obviating the need for the operator Tableopr(C,B). The reason PABLO did not recognize this possibility is precisely because it did not work on the goal On(C,F) until the base level. When the action

Tableopr(y,D) was inserted at abstraction level 1, $On^1_{rel}(C,F)$ was satisfied so there was no need to consider the action Puton(C,F), although it would have resulted in a legal plan at that level of abstraction. This is an example of the trade-off between efficiency and optimality that planning with relaxed predicates introduces.

3.7 Extensions to Predicate Relaxation

3.7.1 Associating Costs with Operators

One way in which predicate relaxation can be extended is to associate costs with operators and define predicate relaxation levels in terms of particular costs and not simply in terms of the number of operators.

Before we elaborate on this idea, it is important to be aware of two distinctions. First, basic predicate relaxation provides a measure of how difficult it will be to *plan* to achieve a certain predicate. It does not provide a measure for how difficult it will actually be for the executor of the plan to achieve that predicate. The reason the former measure is of value to us, is that it is planning time we are trying to minimize. Therefore, it seems reasonable to concentrate on the predicates for which a simple plan does not exist.

The second distinction is that predicate relaxation relaxes predicates over uninstantiated operators, i.e. over the operator templates. Therefore, if we are interested in the cost associated with executing an operator it might not be available. For example, if in our domain we have a *drive* operator, the cost associated with it will not be known until it is instantiated, and might vary considerably. The cost of driving two miles to school is significantly different from driving across the country.

Given these distinctions we can generalize predicate relaxation to relax predicates over operator templates with costs associated with them. For example, in a travel domain, the highest cost might be associated with the *fly* operator and the lowest with the *drive* operator. Then, instead of relaxing predicates in terms of the number of operators necessary to achieve them, we relax in terms of cost threshold. e.g. In the travel domain, at a particular relaxation level, we would allow more *drive* operators in a relaxation of a particular predicate than *fly* operators. This would have the effect of introducing the *fly* operators into the plan at a higher level of abstraction than the *drive* operators.

3.7.2 Limit Relaxation Operators

Another possible extension is to limit the regression of predicates during the relaxation phase to a subset of possible operators, thus creating smaller relaxation definitions. For example, we might limit the relaxation definitions to only include commonly used operators. Also, certain operators might achieve a predicate as a side effect. We might limit the relaxation definitions to only those operators which have a predicate which is being regressed as a main effect. We will see later, in chapter 8 how this and other techniques can be used to significantly reduce the size of the predicate relaxations.

3.7.3 Relaxation over Hierarchical Operators

In chapter 8 we will see how we can extend the PABLO operator representation to include hierarchical operators. It will then be possible to plan using both types of abstractions. We will give an example where it will be desirable to relax predicates over hierarchical operators, so there is no longer a one to one correspondence between the relaxation level and the number of primitive actions over which we regress.

3.8 Conclusion

We have presented PABLO, a non-linear hierarchical planner that automatically generates abstraction spaces using predicate relaxation. PABLO is able to solve some problems, e.g. Towers of Hanoi, making full use of the abstractions inherent in the domain, something which previous planners could not. The resulting abstraction spaces can greatly increase planning efficiency. Predicate relaxation has several advantages over ABSTRIPS's abstraction technique. It is fully automated, it provides a gradual abstraction of predicates, and the number of abstraction levels can be tailored to the particular problem to be solved.

Chapter 4

Complexity Analysis

4.1 Introduction

We might ask what we can gain by using predicate relaxation. Korf [Korf, 1987], and later Knoblock [Knoblock, 1990] have shown that planning with abstractions can reduce worst-case planning complexity from exponential in the length of the resulting plan, to linear in the length of the plan.

In this chapter we review this work, and then show a complexity analysis of planning with predicate relaxation.

4.2 Previous Work

One of the first analyses of abstraction was made by Korf [Korf, 1987]. He was able to show that with a properly constructed abstraction hierarchy it is possible to reduce planning time from exponential to linear complexity in the length of the resulting plan. However, in the context of traditional planning, the construction used by Korf is somewhat non-standard in that it assumes that if there is a path between two states at a high level of abstraction, we automatically know what the path is at a lower level of abstraction. In hierarchical planning this is normally not the case, and we must generally plan at the lower level in order to determine the lower level path. The upper abstraction levels provide the lower levels with islands which guide the planning process.

Knoblock [Knoblock, 1990] has analyzed abstraction in planning without this assumption, showing again that it is possible to reduce worst case planning complexity from exponential to linear in the length of the final plan. In our analysis of planning with relaxed predicates we make use of this result.

4.3 Complexity Analysis

4.3.1 Planning at one level

We will define $P(l)$ to be the worst case complexity of finding a plan consisting of l actions. In the case of a state-space planner $P(l) = \sum_{i=0}^{l} b^i$, where b is the branching factor of the state space. This is the model used by both Korf [Korf, 1987] and Knoblock [Knoblock, 1990]. However, most planners are plan-space planners [Sacerdoti, 1977, Wilkins, 1984, Chapman, 1987], including PABLO. Computing the worst case complexity of a plan-space planner is still an open problem, although it is likely to be at least exponential in the length of the resulting plan.

See figure 4.1 for an illustration of a planner planning hierarchically. The branching factor corresponds to the number of subproblems that each level generates at a lower level. In the figure this branching factor is 2. We refer to this branching factor of the abstraction space as c. If the final plan is of length l, the height of the tree will be $\log_c l$.

The complexity of planning using n levels of abstraction, assuming the complexity of planning does not vary with the abstraction level is

$$P(c) + cP(c) + c^2P(c) + \ldots + c^{\log_c l}P(c)$$

where c is the branching factor of the abstraction space.

However, if we want to compute the complexity of planning with relaxed predicates we need to take into account the greater amount of time it takes to determine the truth of a predicate at a high level of abstraction.

As we relax a predicate from one level to another we disjoin the predicate with its regression through each operator in the domain. There might actually be more than

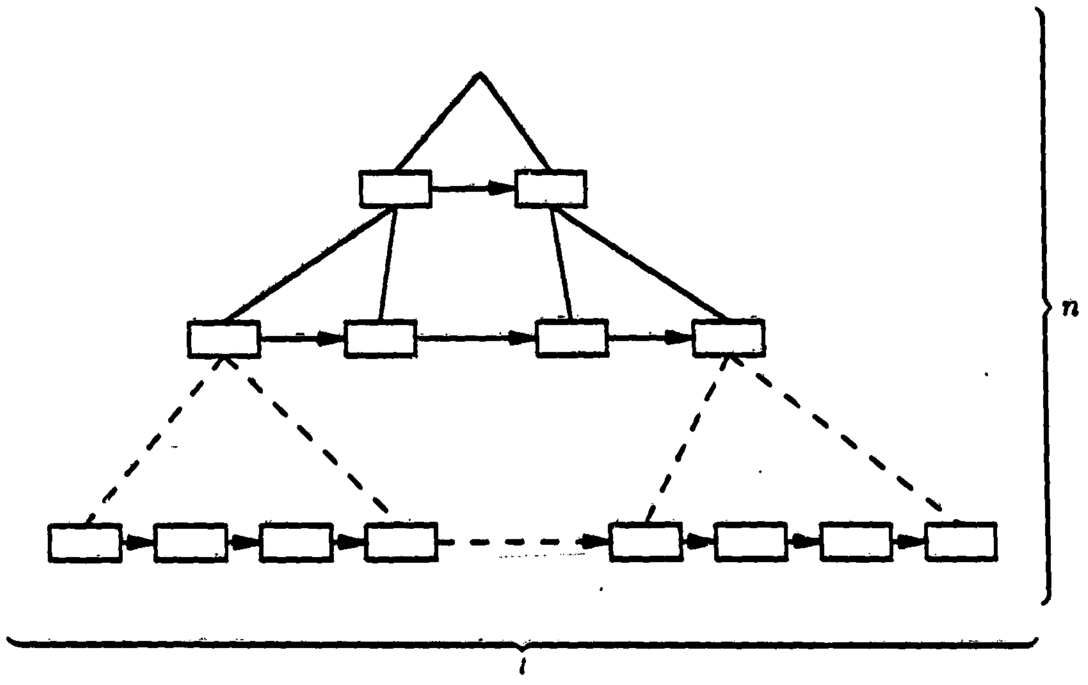


Figure 4.1: Abstraction Space

one regressed expression for a particular operator since the predicate might match more than one predicate in the add list. If we have o operators in the domain, and at most s predicates in the add list, each predicate's regression can consists of at most os conjunctions Therefore, if a particular conjunction consists of p predicates, the resulting regression will consist of at most osp conjunctions.

Notice that as we proceed with the regressions from level to level that the size of the conjunctions will increase, assuming we do not perform simplifications. If we take d to be the maximum size of any precondition of the operators, then the maximum number of predicates at regression level n is bounded by nd.[1] Note that the size of the conjunction might actually be larger since we might generate equality and inequality predicates. However, these are simply passed from level to level and do not generate any new regressed expressions. Therefore, at relaxation level n the number of new conjunctions generated by regressing a conjunction at level $n-1$ is bounded by $os(n-1)d$.[2] If at level $n-1$ we have c_{n-1} conjunctions, at level n we will have at most $os(n-1)dc_{n-1}$ conjunctions. Therefore, at level n, the number of conjunctions is bounded by $(osd(n-1))^n$. Renaming osd to be k we get $(k(n-1))^n$. For simplicity of exposition we will use the weaker bound of $(kn)^n$ which is valid for $n > 0$. For $n = 0$ there are only individual predicates so the bound is 1.

Given that the number of conjunctions in a relaxed predicate is bounded by $(kn)^n$ we must also establish the complexity of computing the truth of each conjunction. We set z to be the maximum number of predicates in a particular situation. In the worst case, we might have to try every possible instantiation of each predicate in a conjunction of length p which takes z^p. Since the maximum number of predicates in a conjunction that need to be checked in a situation (i.e. not the equality and inequality constraints) is bounded by dn at level n this becomes z^{dn}. The equality and inequality constraints can each be checked in constant time and their number is bounded by $O(n^2)$.[3]

[1] In what follows we assume $n > 0$.

[2] This is only valid for $n > 1$

[3] This is because at level n, the number of new equality constraints generated by regressing a conjunction from level $n-1$ is bounded by $fd(n-1)$, where f is the maximum size of the delete list of any operator, and $d(n-1)$ is the maximum number of nonequality predicates in a conjunction at level $n-1$. The expression $\sum_{i=2}^{n} fd(i-1)$ is bounded by kn^2 for some k.

Therefore, the complexity of computing the truth of an abstract predicate at abstraction level n is $O(z^{dn}(kn)^n)$. Simplifying we get $O((z^d kn)^n)$ which is $O((Kn)^n)$ for $K = z^d k$.

Given this, we can now derive an expression for the complexity of planning with relaxed predicates.

$$O\left((Kn)^n P(c) + c(K(n-1))^{n-1} P(c) + \ldots + c^n P(c)\right)$$

We can write the above as

$$O\left(c^n P(c) + \sum_{j=1}^{n} (Kj)^j c^{n-j} P(c)\right)$$

Moving the constants out of the sum we get

$$O\left(P(c)c^n(1 + \sum_{j=1}^{n} (Kj/c)^j)\right)$$

The sum is bounded by $1 + (gn)^n$ for some g.

$$O\left(P(c)c^n(1 + (gn)^n)\right)$$

Simplifying

$$O\left(P(c)(c^n + (cgn)^n)\right)$$

Renaming cg to G and noting that n is $log_c l$ we have

$$O\left(P(c)(c^{log_c l} + (Glog_c l)^{log_c l})\right)$$

Which is equivalent to

$$O\left(P(c)(l + l^{log_c G} l^{log_c log_c l})\right)$$

Which can be simplified to

$$O\left(P(c)(l + l^{log_c G + log_c log_c l})\right)$$

By noting that G and c are constants and renaming $log_c G$ to y this reduces to

$$O\left(l + l^{y + log_c log_c l}\right)$$

The expression $y + log_c log_c l$ grows very slowly and is therefore very close to being a constant, so the complexity has been reduced to *nearly* polynomial in the size of the final plan. It should be noted that if we can guarantee that the number of conjunctions at each abstraction level grows exponentially, as opposed to being bounded by $(kn)^n$, we can reduce the planning complexity to polynomial in the length of the final plan. As we shall see this occurs in several domains in which we test PABLO.

Suppose, for example, that we bound the maximum size of any conjunction in the relaxation expression to e. Then, the maximum number of conjunctions at level n is bounded by $\sum_{i=0}^{n}(ose)^i$, where we recall that o is the number of operators in the domain and s is the maximum size of the add list. The maximum number of conjunctions in a relaxation expression at level n is therefore bounded by $O((ose)^n)$. We refer to the constant ose as m. Determining the truth of a conjunction in a situation with a maximum of z predicates is bounded by z^e which is a constant. Therefore, the complexity of determining the truth of a relaxed predicate at level n is bounded by $O(m^n)$.

Our expression for the cost of planning now becomes

$$O\left(m^n P(c) + cm^{n-1}P(c) + \dots + c^n P(c)\right)$$

We can write the above as

$$O\left(\sum_{j=0}^{n} m^j c^{n-j} P(c)\right)$$

Moving the constants out of the sum we get

$$O\left(P(c)c^n \sum_{j=0}^{n}(m/c)^j\right)$$

Solving the sum we arrive at

$$O\left(P(c)c^n((m/c)^{n+1} - 1)/((m/c) - 1)\right)$$

Simplifying

$$O\left(P(c)(m^{n+1} - c^{n+1})/(m - c)\right)$$

But n is $log_c l$ so we have

$$O\left(P(c)(m^{1+log_c l} - c^{1+loc_c l})/(m - c)\right)$$

Which is equivalent to

$$O\left(P(c)(ml^{log_c m} - cl)/(m - c)\right)$$

Since c is a constant this becomes

$$O\left((ml^{log_c m} - cl)/(m - c)\right)$$

This reduces to $O(l^k)$ where $k = \max(1, \log_c m)$.

Therefore, if we can bound the growth of the relaxed expressions to be exponential in the number of abstraction levels we can reduce planning complexity from exponential to polynomial in the length of the final plan.

4.4 Discussion

As we have seen, it is possible to reduce exponential planning time to nearly polynomial in certain circumstances. However, this is only the case if certain assumptions we have made along the way hold. First, it must be the case that there is no backtracking across abstraction levels. Second, within an abstraction level, there must be no backtracking across the subproblems of that abstraction level. Each subproblem must be solved independently of the others. Third, the length of the final plan of the abstraction planner must be the same as the length of the plan found by the non-abstracting planner. Finally, there must be a uniform branching factor of the abstraction space. If any of these assumptions fail, the analysis no longer holds and the planning reverts to possibly exponential complexity. Note that these are the same assumptions that Knoblock makes in his complexity analysis [Knoblock, 1990].

As we shall see in the next section, in practice these assumptions generally hold fairly well for the domains we have attempted. The more amenable a domain is to being abstracted, the better these assumptions hold.

Furthermore, the complexity of planning grows with the size of y. It is therefore in our interest to reduce the size of the relaxed predicates as much as possible to reduce the time to compute their truth value. This can be done by simplifying, as well as invoking domain constraints to rule out impossible expressions. Even though theoretically the complexity is considerably improved when using relaxed predicates, in practice it is important that they not become unwieldy, since this might result in an impractically large constant in the complexity formula. We will have more to say about this in future chapters.

Chapter 5

Empirical Analysis

5.1 Introduction

In this chapter we present empirical results of applying PABLO to four domains. In each of the domains we compare the performance of PABLO without relaxed predicates to PABLO with relaxed predicates. The domains we test PABLO in are Towers of Hanoi, Blocks World, Robot World, and the Eight Puzzle.

From our theoretical analysis we should expect potential gains in planning efficiency when applying predicate relaxation. As we shall show, this is indeed the case.

The data presented in this chapter are from an implementation of PABLO on a Symbolics 3620, under Genera 7.1. It should be noted that very little optimization was performed on PABLO. The numbers should serve as a means of evaluating the usefulness of predicate relaxation, not as a testament to the ultimate speed of planning.

5.2 Towers of Hanoi

To show the power of using predicate relaxation in the Towers of Hanoi domain, we generated seven problems in the 3 disk problem, each successive problem requiring a solution with a length of one more operator than the previous one. We ran the

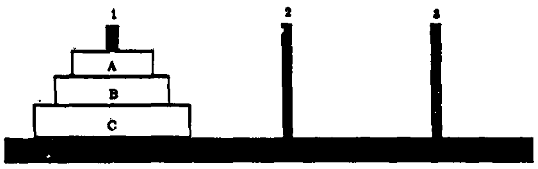


Figure 5.1: Towers of Hanoi

problems on PABLO with predicate relaxation and without. A plot of the respective planning times can be seen in figure 5.2.

As can be seen from the graph, the time it take PABLO to find a solution when using predicate relaxation grows linearly with the number of operators in the plan. Without the use of predicate relaxation the running time grows exponentially. These results conform to our theoretical analysis of predicate relaxation. Notice that each of the assumptions of the complexity analysis holds in this example: no backtracking across abstraction levels; no backtracking across subproblems; the optimal solution is generated; and a uniform abstraction-space branching factor. This results in the utility of the relaxed predicates being maximized in this example.

The Towers of Hanoi is in some sense the canonical example for testing reasoning with abstraction, and the gains seen therein are therefore unusually large. Any system that reasons with abstractions should be able to show similar gains for the Towers of Hanoi.

5.3 Blocks World

PABLO was tested on every distinct four-blocks problem. The optimal solution length of the problems range from 1 to 6 steps. We ordered the problems according to the optimal solution length, and then averaged the time to solve the problems of each length. This was done both with the use of relaxed predicates, as well as without their use. The results are presented in figure 5.3.

As can be seen we see significant speedups in the blocks world as well. It should be noted that the optimal solution was not always discovered when using the relaxed

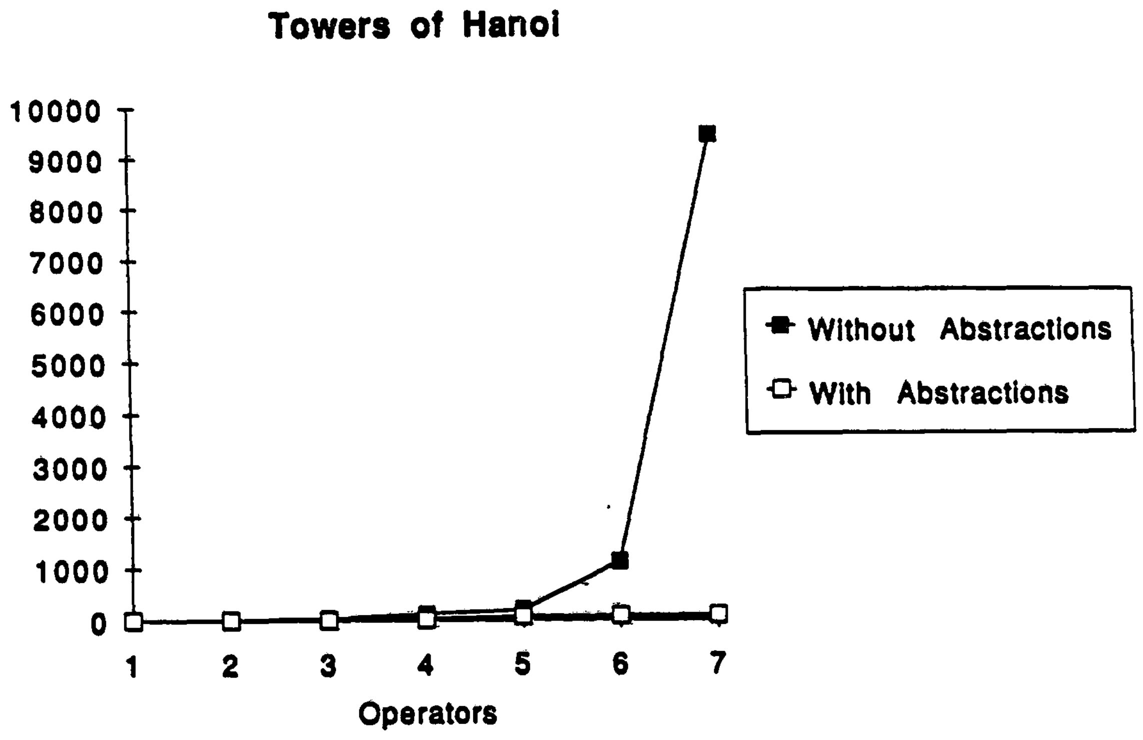


Figure 5.2: Running times of PABLO for the Towers of Hanoi (in seconds).

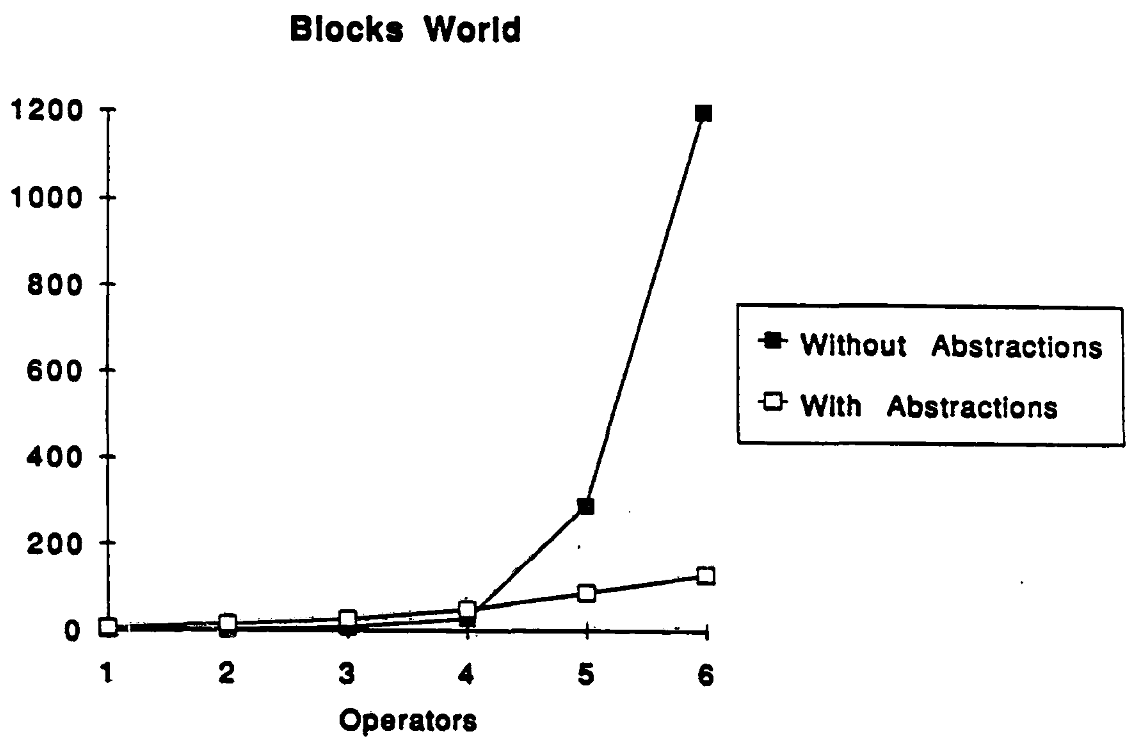


Figure 5.3: Running times of PABLO for the Blocks World (in seconds).

predicates. Out of 223 problems, PABLO generated plans that were one step longer than optimal when using abstractions 15 times. No other suboptimal plans were generated. As we have pointed out earlier this is one of the tradeoffs that is made when using abstractions, namely optimality for efficiency. In general, we believe this to be a worthwhile tradeoff.

The gains in the blocks world were not as spectacular as those in the Towers of Hanoi. This is because the blocks world is not as amenable to abstraction as the Towers of Hanoi. It is interesting though that significant gains were still observed.

5.4 Robot Domain

The third domain tried was the robot world domain similar to that used by STRIPS and ABSTRIPS. See figure 5.4 for a typical example. A problem in this domain might involve moving the robot from room G to room A and also push two boxes next to each other. A plan for solving this problem might involve opening doors and pushing boxes from one room into another. Typical operators for this domain can be found in chapter 8.

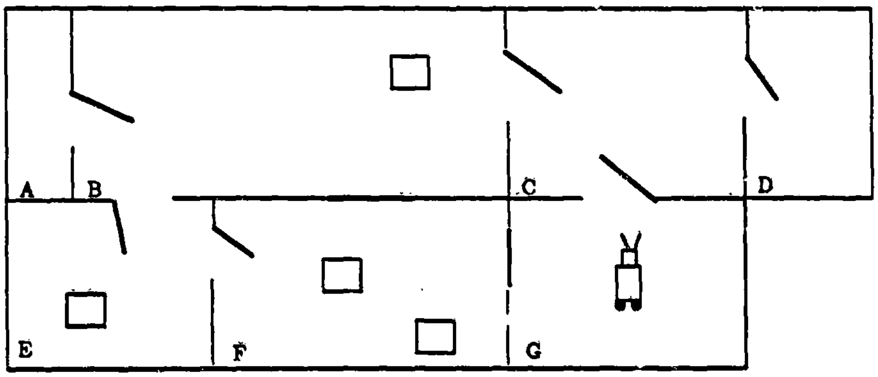


Figure 5.4: Robot World Domain

Random problems were generated and ordered according to their optimal solution length, as in the blocks world. The times to solve problems of each solution length

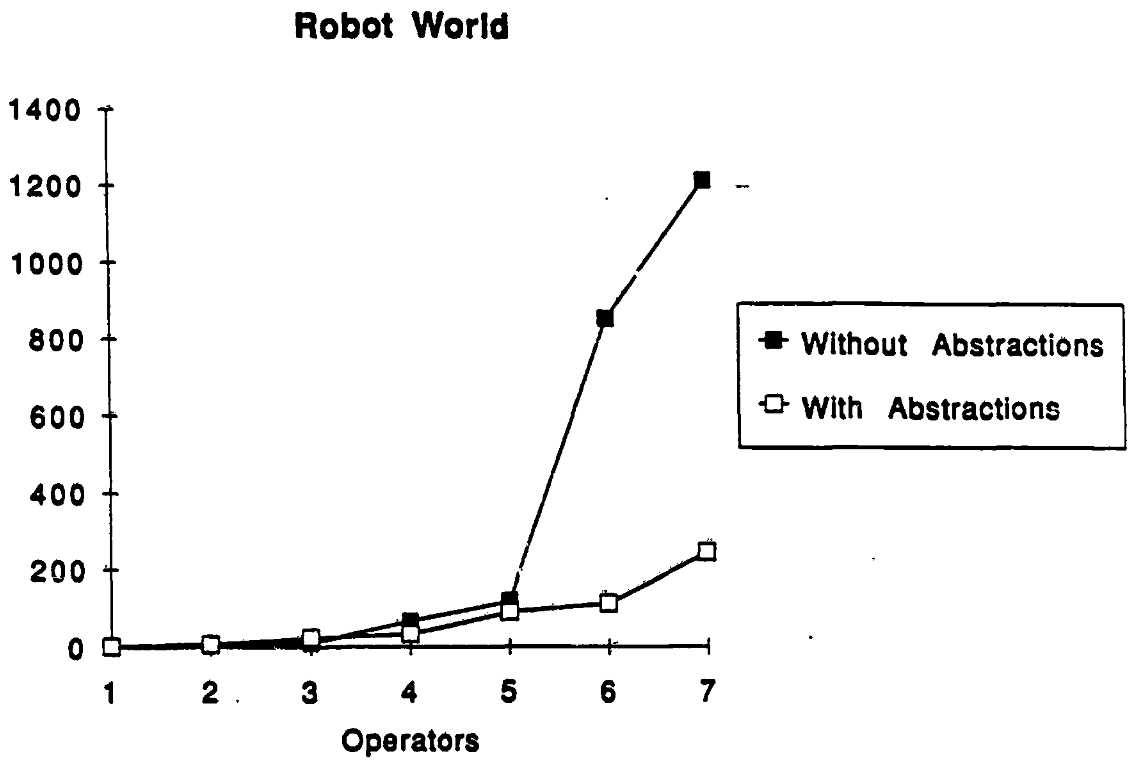


Figure 5.5: Running times of PABLO for the Robot World (in seconds).

were averaged with and without using predicate relaxation. The results can be seen in figure 5.5.

5.5 Eight Puzzle

The fourth and final domain in which PABLO was tested was the eight puzzle. This puzzle entails sliding eight tiles on a square grid, where there are nine locations. Figure 5.6 shows a typical initial state and goal configuration for the eight puzzle.

As in the previous two domains we ordered the problems according to the optimal solution length and averaged the times to solve the problem with relaxed predicates and without. The results can be seen in figure 5.7.

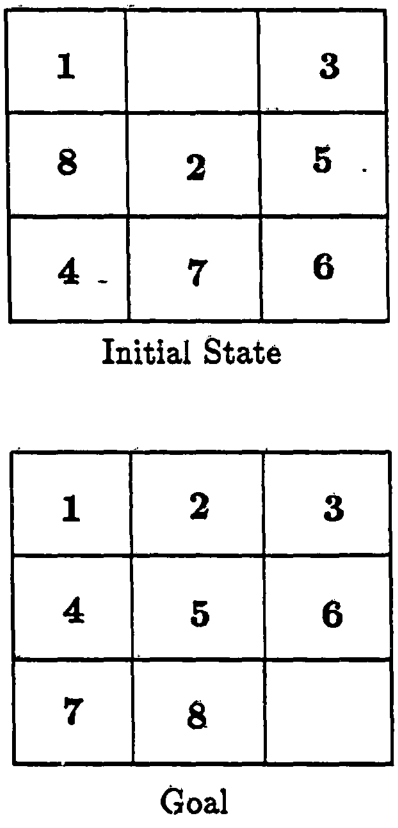


Figure 5.6: Eight Puzzle

5.6 Discussion

As can be seen from the empirical results, PABLO improved its performance significantly with the use of relaxed predicates. The only drawback was a slight overhead on easier problems, and the possibility of suboptimal solutions.

It should be noted that all the domains in which we have tested PABLO are so-called toy domains, i.e. they are unrealistically simple. The reason this was necessary is that the underlying planning algorithm used by PABLO, due to its completeness, is inherently slow, and ill suited to real world tasks. As we have explained earlier, there are good reasons for using an underlying algorithm that is complete when experimenting with abstractions. With a complete planner it is much easier to gauge the effects introducing abstractions has on the planner. It is of great interest to see if the ideas scale up to real world problems.

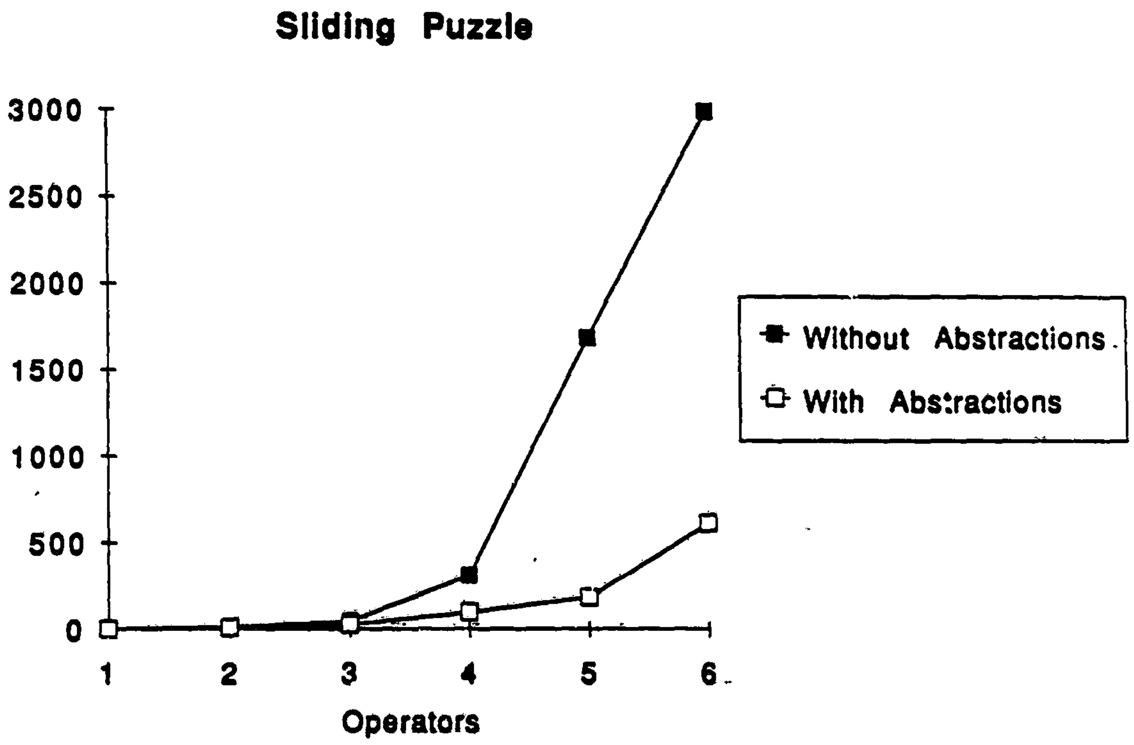


Figure 5.7: Running times of PABLO for the Eight Puzzle (in seconds).

Chapter 6

Using Abstractions to Achieve Reactivity

6.1 Introduction

There is a growing body of research concerned with tackling the problem of planning in unpredictable, uncertain, and time stressed environments. One of the oft cited shortcomings of the classical planning approach is that it assumes a benign environment, of which the planner has complete knowledge, nothing untoward happens during execution and the planner has virtually infinite time in which to plan. As a result, most of the new approaches have eschewed classical techniques in favor of more radical approaches. We will present a brief overview of some of the main techniques proposed to deal with more complex environments.

6.1.1 Universal Plans

Schoppers [Schoppers, 1987] has proposed that rather than plan, an agent should use *Universal Plans* in unpredictable, time stressed environments. A Universal Plan is a function from sensor inputs to actions which computes the appropriate action to perform in a situation. Thus, the agent has a precomputed action for every possible situation it can find itself in. One way to store such a universal plan is as a decision tree. Clearly, having such an action cache guarantees very fast response time. However, as we shall discuss later there are some problems with it.

```
on(a,b) ?
   T) at(top) ?
   .      T) NO-OP
   .      F) ¬holding(a) ?
   .             T) RAISE
   .             F) OPEN
   F) clear(b) ?
          T) holding(a) ?
          .      T) over(b) ?
          .      .      T) LOWER
          .      .      F) at(top) ?
          .      .             T) LATERAL
          .      .             F) RAISE
          .      F) [subplan to GRASP a]
          F) [subplan to CLEAROFF b]
```

Figure 6.1: A piece of a Universal Plan represented as a decision tree

In figure 6.1 we see a Universal Plan for guaranteeing that On(a,b) holds. The plan is represented as a decision tree. Each condition in the tree is tested until a leaf is reached. The leaf specifies the appropriate action to perform.

6.1.2 Action Nets

Nilsson has proposed Action Nets as an architecture in which to implement reactive systems [Nilsson et al, 1990]. An action net is a network of units, each of which has some inputs (preconditions, a trigger, and a goal), and one output. The output is connected to another unit or to a switch of some kind which activates an action in the external world. As with the previous approaches an action network guarantees very fast response times. Furthermore, there are facilities for dynamically expanding the network at run-time, a feature other approaches lack. But, to date, it is not clear how action nets might interface with an automatic planning system.

6.1.3 Situated Action Rules

Agre and Chapman [Agre and Chapman, 1987] have developed Pengi, a program designed to play the Pengo video game. This game requires quick response times. Pengi, unlike a Universal Plan, is not purely functional, but retains some state in its "vision" system. Even with relatively simple rules, quite complex playing behaviour is achieved by Pengi. For a discussion of the differences between Pengi and other reactive approaches see [Chapman, 1989].

6.1.4 Subsumption Architecture

Brooks [Brooks, 1986] proposes the *subsumption architecture* for controlling an agent. The main idea in this proposal is to organize the agent vertically according to levels of task-behaviors, with higher levels performing more complex tasks. When a higher level completes its computation it can subsume all lower levels. Presumably, higher level computations are on the average more time consuming. When pressed for time, the agent uses the result of its lowest level behaviors. However, if given more time, one of the higher levels can subsume the lower levels with an action of higher quality.

This is by no means an exhaustive listing of the approaches proposed for reactivity. Other interesting ones include [Rosenschein and Kaelbling, 1987, Firby, 1987, Drummond and Currie, 1988].

6.1.5 Discussion

We will take the liberty of referring to the above approaches as *reactive planners*. Ginsberg [Ginsberg, 1989] points out several serious problems with reactive plans. The most serious of the problems is that the size of reactive plans grows exponentially with the size of the domain. Although it remains to be shown, it is likely that the domains which AI concerns itself with will be complex enough that reactive plans will grow prohibitively large.

It seems clear that some amount of run-time inference is necessary for any agent to act successfully in an environment. However, it is also necessary to provide some mechanisms for reactivity, for situations where there simply is no time for complex deductions.

6.1.6 Classical Approaches to Reactivity

Suppose a planner is given the following problem to solve (see figure 6.2 for an illustration of the problem.)

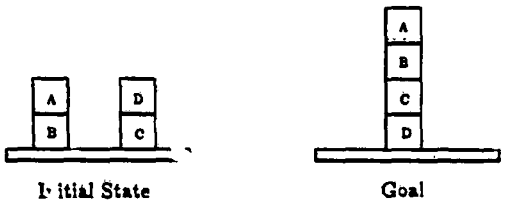


Figure 6.2: Planning Problem

We are given the following two operators:

PUTON(x,y)	TABLEOPR(x)
P:{Clear(x),Clear(y),On(x,z)}	P:{Clear(x),On(x,y)}
D:{Clear(y),On(x,z)}	D:{On(x,y)}
A:{Clear(z),On(x,y)}	A:{Clear(y),On(x,TABLE)}

Using the classic non-linear planning method a trace of the plan at various stages of development might look as in figure 6.3.

One notable feature of this trace is that until the final plan is produced the classical planner is not aware of any executable actions to perform in the initial state. The actions Puton(B,C) and Puton(C,D) are not directly executable in our initial state. Should the planner be interrupted at any time during planning with the need to start executing immediately it would not have a reasonable action to perform.

The problem is that the classical planning approaches have invariably been backward chaining. This is a perfectly reasonable strategy given that in most planning domains the branching factor is considerably reduced when reasoning from the goal back to the initial state. However, it has the drawback that until the final plan has been developed, there is no guarantee that the plan will actually be applicable in the initial state.

This is one reason traditional planning methods have generally been regarded

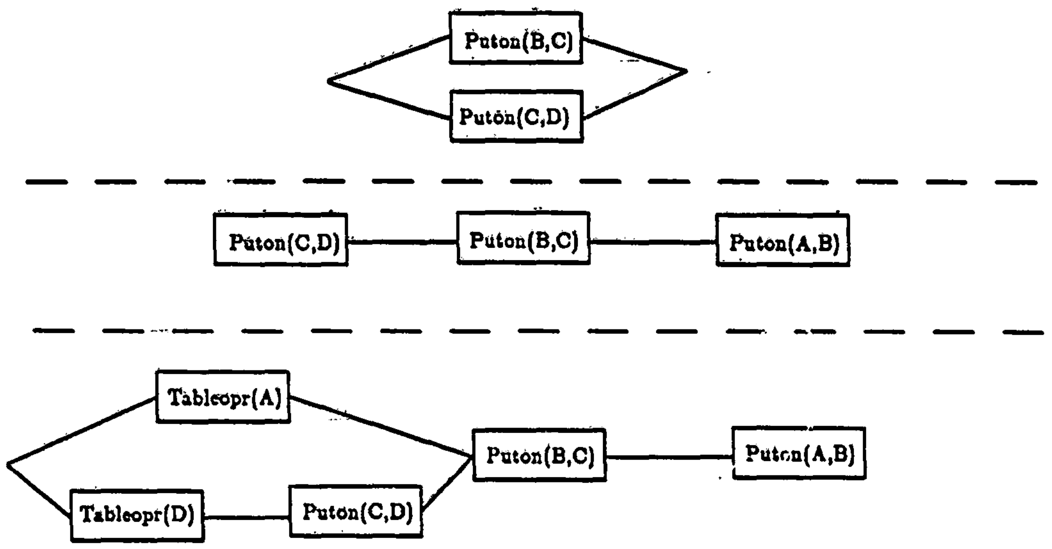


Figure 6.3: Classic Planning Trace

as unsuitable for real-time tasks. Besides the reactive planning methods mentioned earlier, several other alternatives within the classical planning context have been suggested.

6.1.7 Forward Search

Some favor abandoning backward chaining plan-space search in favor of a forward search of the state space [Washington, 1989]. The advantage of this approach is that an executable action is available as soon as an action has been found applicable in the initial state. Unfortunately, until we encounter the final solution during the forward search we have no guarantee that our current sequence of actions will eventually lead to the goal. Further, we cannot take advantage of the least-commitment implicit in the non-linear representation of plans. Finally, a forward search, so as not to be completely blind, needs a domain-specific heuristic, thereby reducing domain-independence.

6.1.8 Left Recursive Wedge Planning

Another approach one can take is that of planning down a *left-recursive wedge* of the partial plan in case of an interruption [Wilkins, 1988]. The idea is to repeatedly expand the leftmost outstanding preconditions until an action is encountered with all its preconditions satisfied. In some circumstances this approach might be successful. Unfortunately, the time to plan in this manner is possibly unbounded, since interactions might be encountered that necessitate backtracking.

The method we propose retains the power of partially ordered plan representations, but also allows the planner to identify plausible executable actions early in the planning process.

6.2 Reactive Reasoning with PABLO

The problem we are addressing is that of providing a plausible executable action should PABLO be interrupted before it has formed a complete plan. Ideally, we would like to provide as long a sequence of executable actions as possible.

Toward this end, we can store, along with each relaxed predicate, the operators through which the predicate was regressed during the relaxation process. Then, during planning, when a relaxed predicate is determined to hold in a situation, the operator through which the predicate was last regressed is automatically identified. e.g. this is the first level relaxation of On(x,y):

$On^1_{rel}(x,y)$	
On(x,y)	[]
(y = TABLE) ∧ Clear(x) ∧ ∃ z On(x,z)	[Tableopr(x)]
(y ≠ TABLE) ∧ Clear(x) ∧ Clear(y) ∧ ∃ z On(x,z)	[Puton(x,y)]

The logical expressions are conditions under which the predicate should be determined to hold. The operators through which the predicate was regressed to arrive at the expression are shown in the right side of the table. In this case, since it is a first level relaxation, only sequences having one operator are included.

During planning, the relaxation table is examined from top to bottom. When an expression is found that is satisfied in the current state, the relaxed predicate is said to be asserted in that state. We also say the relaxed predicate is *grounded* in this state. This is important because a relaxed predicate that is grounded in a state must have a sequence of operators that is executable in that state, which guarantee that the predicate with hold as a result of their execution.

6.3 Identifying Executable Actions

Once PABLO has completed a plan at one level of abstraction, and is working at the next lower level, it can utilize the extra information stored along with the relaxed predicates that hold at the higher level, should it be interrupted.

PABLO chooses a plausible action by examining the preconditions of the earliest action(s) of the plan. If one of these actions has all its preconditions satisfied at the base level, the action is obviously executable.

If no such action exists, PABLO can choose from among the leftmost operators associated with the satisfied predicate relaxations that are grounded in the initial situation. All relaxed predicates must be satisfied since the plan was completed at the higher level. Furthermore, at least one of the relaxed predicates must be grounded in the initial state. To see this note that there must be at least one action such that no action is necessarily between it and the final situation. If all its preconditions were satisfied at the base level the action itself would be executable. If not, every predicate that is abstractly satisfied in the precondition must be grounded in the initial state. Any of the actions collected in this manner are executable.

See figure 6.4 for a trace of PABLO solving the previous example. It should be noted that PABLO solves the example twice as fast using the predicate relaxations than without, since reasoning with the abstractions allows it to prune substantial portions of the search space.

After completing planning at the second level of abstraction the plan consists of one operator: Puton(B,C). This is because all preconditions of Puton(B,C) are satisfied at this level of abstraction, and because the remaining goals, On(A,B) and

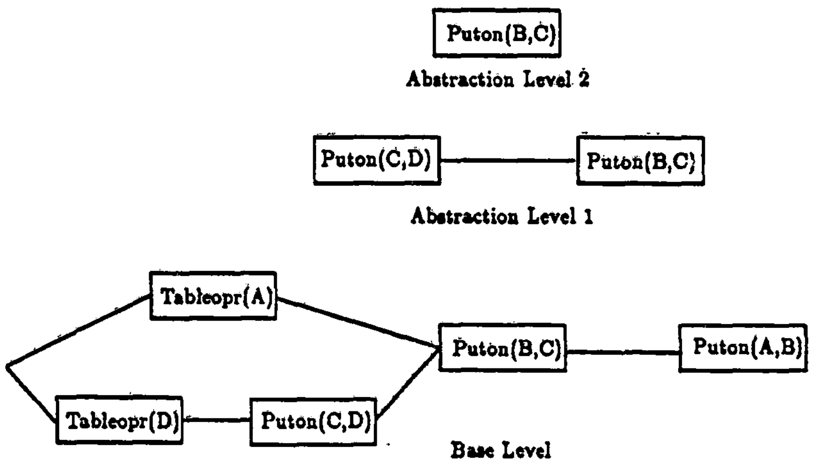


Figure 6.4: Pablo's Planning Trace

On(C,D) are also satisfied (at this level of abstraction). See figure 6.5 for a detailed description of the planning state at this point.

As PABLO moves down to the first abstraction level, the goal On(C,D) is no longer satisfied since it requires two steps to accomplish. PABLO grows the plan by adding the operator Puton(C,D) to achieve this goal. Notice that at the first level of abstraction all preconditions to Puton(C,D) hold, since D is clear and block C can be cleared in one step. PABLO then completes the plan at the base level.

Now, suppose PABLO is interrupted after it has completed planning at abstraction level 2. At this level there are three predicates that hold abstractly, i.e. the components of their relaxed definitions that are satisfied have non-null operator lists associated with them. These are $\text{Clear}^2_{\text{rel}}(B)$, $\text{Clear}^2_{\text{rel}}(C)$, and $\text{On}^2_{\text{rel}}(C,D)$.

To see this, examine the second level predicate relaxation of Clear(x).

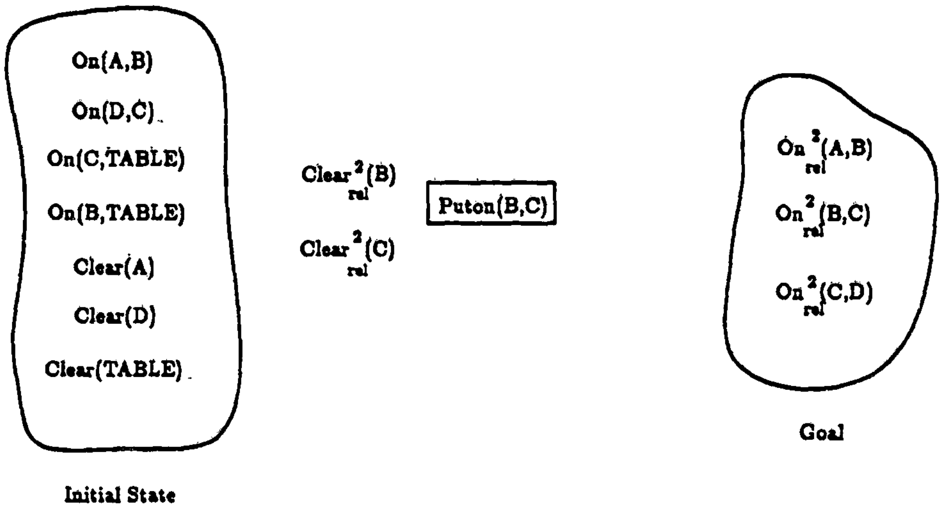


Figure 6.5: Level 2 Plan

$\text{Clear}^2_{\text{rel}}(x)$	
Clear(x)	[]
∃ y On(y,x) ∧ Clear(y)	[Tableopr(y)]
∃ y,z On(y,z) ∧ Clear(y) ∧ On(z,x)	[Tableopr(y),Tableopr(z)]

The above table has been simplified by removing subsumed expressions. e.g. the result of regressing Clear(x) through Puton(y,z) is

$$\exists y, z\ \text{On}(y, x) \wedge \text{Clear}(y) \wedge \text{Clear}(z)$$

This expression is not included in the table since it is subsumed by the regression of Clear(x) through Tableopr(y). When $\text{Clear}^2_{\text{rel}}(x)$ is instantiated in our plan, with variable x bound to C, $\text{Clear}^2_{\text{rel}}(C)$ becomes:

$\text{Clear}^2_{\text{rel}}(C)$	
Clear(C)	[]
On(D,C) ∧ Clear(D)	**[Tableopr(D)]**
∃ y,z On(y,z) ∧ Clear(y) ∧ On(z,C)	[Tableopr(y),Tableopr(z)]

As can be seen from the predicate relaxation definition $Clear^2_{rel}(C)$ holds because On(D,C) $\wedge$ Clear(D) is true at the base level, and the leftmost regressed operator associated with this regressed expression is Tableopr(D). In brief, the three relaxed predicates hold in the initial state for the following reasons:

$\mathrm{Clear}^2_{\mathrm{rel}}(\mathrm{C})$	On(D,C) $\wedge$ Clear(D)	[Tableopr(D)]
$\mathrm{Clear}^2_{\mathrm{rel}}(\mathrm{B})$	On(A,B) $\wedge$ Clear(A)	[Tableopr(A)]
$\mathrm{On}^2_{\mathrm{rel}}(\mathrm{C,D})$	Clear(D) $\wedge$ On(D,C)	[Tableopr(D), Puton(C,D)]

The above relaxed predicates are grounded in the initial state. If PABLO is interrupted after having completed planning at abstraction level 2, it can choose from among the identified action sequences that are executable in the initial state. In this case it can propose executing Tableopr(D), Tableopr(A), or Tableopr(D) - Puton(C,D). This can be determined as soon as the first level of abstraction has been completed - early in the planning process. In this example it is done after only 15% of the total planning time.

6.3.1 Constructing Incomplete Plans

If necessary, PABLO can construct a substantial portion of the plan, even at this early stage. The outline of the algorithm is as follows:

1. current-plan $\leftarrow$ Nil.

2. current-state $\leftarrow$ $S_{initial}$.

3. op-lists $\leftarrow$ set of lists of operators associated with the predicates that are abstractly satisfied in the plan.

4. op-sequence $\leftarrow$ longest tail of the lists in op-lists that is executable in the current state.

5. if op-sequence = Nil then op-sequence $\leftarrow$ any action in the plan that is executable. If no such action exists, break, returning current-plan.

6. op-lists ← oplists - op-sequence.

7. current-plan ← current-plan | op-sequence.

8. current-state ← S_{out} of last action in op-sequence.

9. Goto step 4.

This algorithm will produce a linear sequence of actions to execute in the current state. See figure 6.6 for the incomplete plan constructed using this technique.

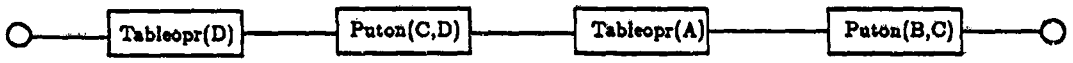


Figure 6.6: Incomplete Plan

The plan is almost complete, the only remaining action is Puton(A,B). Once PABLO commits to the first portion of the plan, developing the remaining portion can be considerably easier.

In this case there is little interaction among the executable alternatives. Therefore, the order in which the algorithm places the operator sequences does not matter, and any of the resulting sequences will be a subsequence of a complete, linear plan for the problem. However, there are obviously cases where such interactions exist.

For example, given Sussman's anomaly, presented in figure 6.7 [Sussman, 1973], the plan, after planning at the first level of abstraction has been completed, will consist of Puton(A,B). There are two operator sequences associated with the two predicates that are abstractly asserted in the plan at this level. The first is Tableopr(C) associated with $\text{Clear}^1_{\text{rel}}(A)$. The second is Puton(B,C) associated with $\text{On}^1_{\text{rel}}(B,C)$.

These are both executable in the initial state, and the algorithm has no a priori reason for choosing between them. Therefore, it might choose to first insert Puton(B,C)

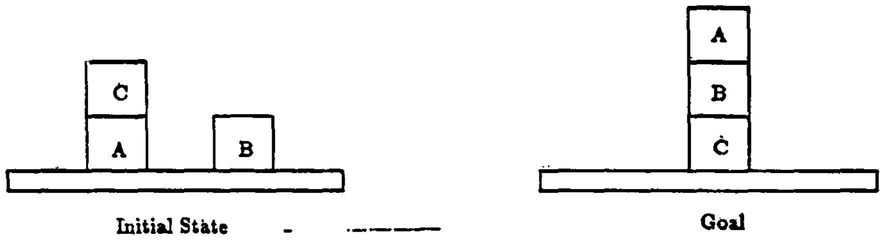


Figure 6.7: Sussman's Anomaly

into the plan, after which no more actions are possible. If it inserts Tableopr(C) first, then the next sequence inserted will be Puton(B,C), which will be followed by Puton(A,B).

Of course, the only way to discover and resolve conflicts based on such interactions is to continue planning. Until a complete plan is produced, we cannot guarantee that the optimal action will be chosen by PABLO (or any other planner), should it be interrupted. This technique, as opposed to the traditional planning algorithm, produces viable alternatives early on in the planning process. Given more time, PABLO will complete plans at succeedingly lower levels, thereby resolving conflicts not discovered at higher levels, and so producing more reliable answers. Our method, in effect, provides a primitive *anytime* algorithm for planning [Dean and Boddy, 1988]

6.3.2 Comparison With other Classical Approaches

Unlike a forward search of the state space, PABLO can take advantage of the least-commitment implicit in non-linear plans. Rather than being committed to one path in the state space at any one time, the partial order of actions represents a set of possible paths that are currently valid. In NOAH, the non-linear representation was found to be successful enough that no backtracking was deemed necessary.

Furthermore, when it is interrupted, its choice for a plausible executable action is derived from a complete abstract plan, which provides a global constraint on this action. An interrupted forward state-space search on the other hand, can only provide local constraints on its choice of executable actions.

Unlike the technique of continuing planning down the leftmost wedge of the plan after an interruption, our approach requires only a bounded computation time to produce an executable action after an interruption. To see this, note that the executable actions are automatically identified after planning at the highest level of abstraction has been completed.

6.4 Comparison to Reactive Plans

The trace of the relaxation of a predicate can be thought of as a reactive plan for achieving that predicate. See figure 6.8 for an illustration of the definition of the $\mathrm{On}^2_{\mathrm{rel}}(x, y)$ predicate as a reactive plan. Notice that some predicates in the reactive plan are not further regressed, this is because these are preconditions to the operator that we do not wish to plan to achieve, but rather just check that they hold. These predicates are specified in the operator definitions given to PABLO. During planning, when a relaxed predicate is determined to hold, the path through the reactive plan that will lead to the establishment of the predicate, is automatically identified.

Our technique is a method for handling these small reactive plans. We believe that this is a more promising approach to reactivity than constructing large, unwieldy reactive plans which risk succumbing to space restrictions very quickly. Each individual plan is restricted in size and can be reused by the planner on different instantiations of the same predicate.

Each reactive plan in our system has a clear purpose, namely to achieve a particular predicate. Unlike other reactive planning techniques which must construct a new reactive plan for each combination of goals encountered (modulo some parameters to the reactive plan), PABLO can re-utilize the reactive plan definitions for any goals specified in the domain.

If our domain is large enough we risk creating abstraction definitions that are too large, although they will always be considerably smaller than reactive plans created for entire domains, since we are only considering reactive plans for individual predicates.

With PABLO, we can extend the planning method to restricted reactive plans, e.g. allow only commonly encountered conditions in the relaxed definitions. Although

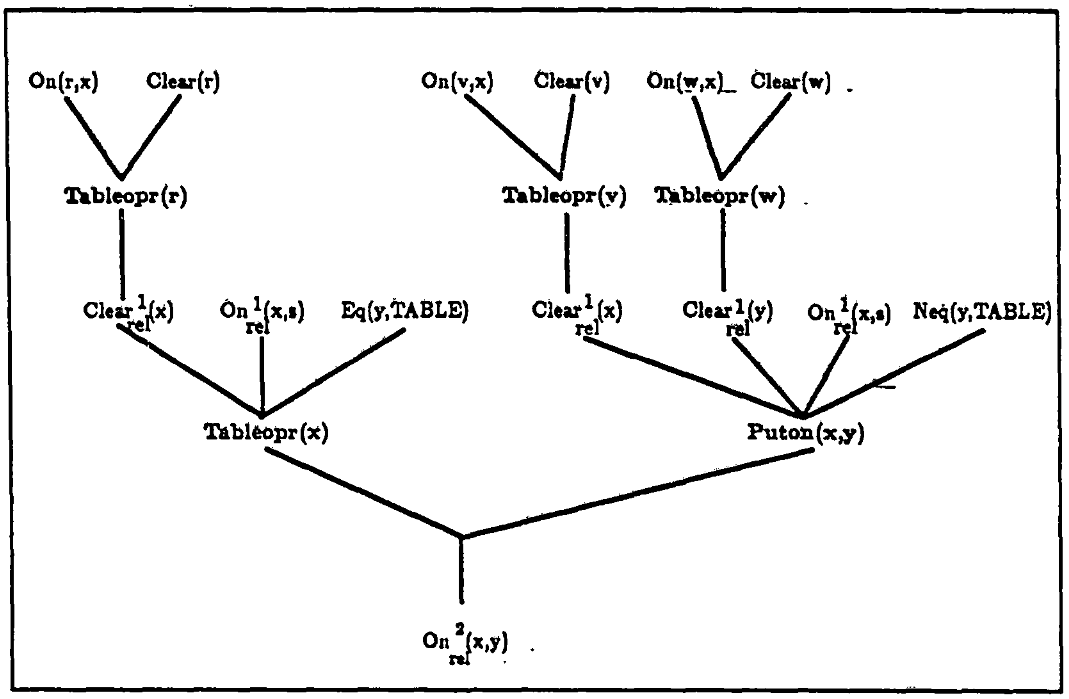

On(r,x)
Clear(r)
Tableopr(r)
On(v,x)
Clear(v)
On(w,x)
Clear(w)
Tableopr(v)
Tableopr(w)
$\text{Clear}^1_{\text{rel}}(x)$
$\text{On}^1_{\text{rel}}(x,s)$
Eq(y,TABLE)
$\text{Clear}^1_{\text{rel}}(x)$
$\text{Clear}^1_{\text{rel}}(y)$
$\text{On}^1_{\text{rel}}(x,s)$
Neq(y,TABLE)
Tableopr(x)
Puton(x,y)
$\text{On}^2_{\text{rel}}(x,y)$

Figure 6.8: Reactive plan for $\text{On}^2_{\text{rel}}(x, y)$

this reduces the number of abstractions identified at the higher levels, each predicate can be more quickly identified to hold abstractly. PABLO is robust in the sense that if a predicate is not deemed to hold abstractly, it can plan to achieve it. This is something systems which rely solely on reactive plans cannot do.

6.5 Conclusion

Using the method presented in this chapter, we can utilize predicate relaxations to produce a plausible executable action should PABLO be interrupted before the final plan has been completed. The only requirement for identifying a plausible action is that planning have been completed at the highest abstraction level. This happens early-in the planning process.

The method can be viewed as an *anytime* algorithm for planning. During planning, harmful interactions are identified and resolved as PABLO plans at succeedingly lower levels of abstraction, thus increasing the quality of the response in case of interruption.

Each automatically generated abstraction is actually a small reactive plan for achieving that predicate. PABLO provides a mechanism for combining these small reactive plans dynamically. Besides the inherent advantages of reasoning abstractly, we can also achieve some measure of reactive behavior, should the planner be interrupted during planning. We believe this is a more promising approach to reactivity than the reasoning with large, unwieldy reactive plans.

Chapter 7

Operator Hierarchicalization

7.1 Introduction

The abstraction of operators has been a prevalent theme in the history of planning. Researchers early realized the value of being able to define abstract operators in terms of other, less abstract, operators. Doing so allows the planner to "jump" from one part of the state space to another with one step, potentially bypassing much planning. However there have been some problems with the proposed abstraction representations which we hope will become apparent.

7.1.1 MACROPS

The first use of what we will label "hierarchical operators" is in STRIPS, with the MACROPS extension [Fikes and Nilsson, 1971]. As the name suggests, this extension allowed STRIPS to learn and use macro operators, for significant computation time savings.

STRIPS would store a plan in a *triangle table* and then apply a procedure to "lift" a generalized MACROP from the triangle table.

MACROPS is interesting for several reasons. It is the first use of hierarchical operators in planning, and it is an example of a planner learning abstractions to speed up problem solving. An example is shown in figure 7.1.

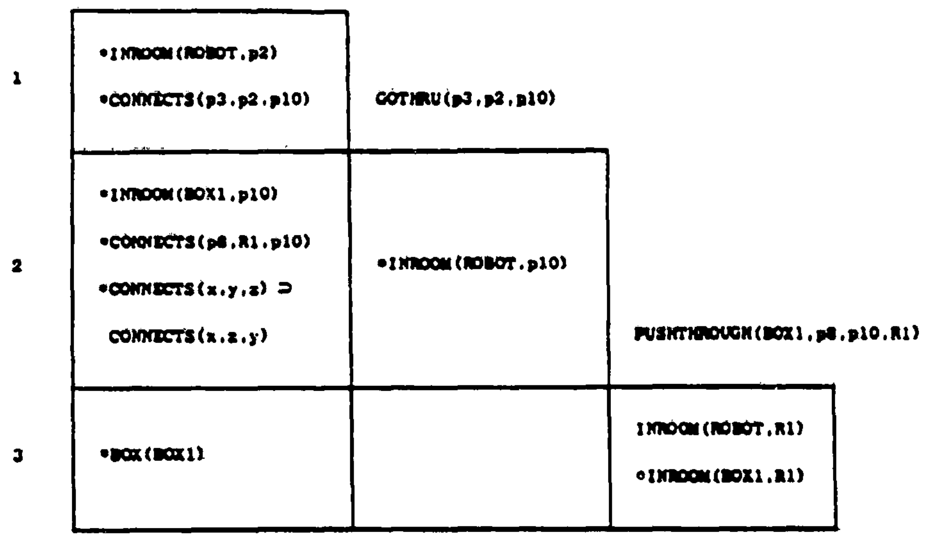


Figure 7.1: A STRIPS MACROP

7.1.2 SOUP operators

With NOAH came the next major advance in hierarchical operators. Operators in NOAH were defined in SOUP (Semantics of User's Problem) code, which allowed for quite general operator definitions. With NOAH came also the term *hierarchical* planning. See figure 7.2 for an example of a NOAH operator.

The SOUP code in the body of the operator provided instructions to NOAH for how an operator should be expanded to the next level of detail. As has been pointed out by several researchers, a NOAH operator is not necessarily hierarchical. For example, in the blocks world, each operator is defined only in terms of a primitive operator and its preconditions. When combined with the NOAH methodology of producing a plan by expanding all the expandable operators in a plan and then critiquing it, a particular ordering in the expansion of goals was imposed.

There is one major drawback with defining operators in this manner. Sacerdoti states [Sacerdoti, 1977]:

```
(PUTON
  (QLAMBDA
    (ON ←X ←Y)
                                          (* Clear $X and $Y, then put
                                          $X on $Y)
    (PAND
      (PGOAL (Clear $X)
             (CLEARTOP $X)
             APPLY
             (CLEAR))
      (PGOAL (Clear $Y)
             (CLEARTOP $Y)
             APPLY
             (CLEAR)))
    (PGOAL (Put $X on top of $Y)
           (ON $X $Y)
           APPLY NIL)
    (PDENY (CLEARTOP $Y))))
```

Figure 7.2: A NOAH operator

The most serious deficiency in the current system is its lack of awareness about the auxiliary computations specified in the procedural semantics (the SOUP code) of a task domain. The procedural net representation lets the system be aware of the goals and subgoals that the planner has decided to tackle, but it does not preserve any information about the computation that resulted in those decisions.

Because the semantics of NOAH operators are opaque to NOAH their usefulness is limited. Primarily, the operators must be defined by the user; it would be very difficult for NOAH to generate new ones. Furthermore, only a minimal amount of error checking can be done by NOAH, which leaves an additional burden on the user of the system.

7.1.3 Procedural Net Operators

The state of the art in hierarchical operators is found in SIPE [Wilkins, 1988]. SIPE solves the problem of semantic opaqueness by defining the plots of hierarchical operators in the same language as that used for its internal procedural net representation. This provides a powerful language in which to define hierarchical operators.

```
Operator: Puton
Arguments: block1, object1 Is Not block1;
Purpose: (On block1 object1);
Plot:
Parallel
    Branch 1:
        Goals: (Clear object1);
    Branch 2:
        Goals: (Clear block1);
End Parallel

Process
Action: Puton.Primitive;
Arguments: block1,object1;
Resources: block1;
Effects: (On block1 object1);

End Plot End Operator
```

Figure 7.3: A SIPE operator

Defining hierarchical operators in terms of a procedural net facilitates not only the design of the planner, in that another language need not be added on top of the existing plan language, but also error checking and the learning of new operators. Although SIPE does not currently learn new operators, its operator representation language provides the infrastructure for transferring knowledge in plan form into operator form.

7.1.4 Formalized Reduction Schemata

In his thesis [Yang, 1989], Yang formalized a version of hierarchical operators similar to those found in SIPE. He defines action templates, which consist simply of preconditions and effects. He then defines a set of action reduction schemata each of which is a function that takes an action template as input and returns a set of partially ordered action templates, with protection intervals between them. The reduction schemata are analogous to plots in SIPE operators.

7.1.5 Problems with Hierarchical Operators

Incorrect Specification of Operators

One problem with the current definition of hierarchical operators is that they might be incompletely or inaccurately specified, i.e. their preconditions and effects might not reflect the actual preconditions or effects of the operator after it has been expanded. One possibility is that the user simply encodes the wrong effects in the postconditions of an operator.. e.g. the encoder of the domain might include a proposition in the add list of a hierarchical operator that is not added by any action in its expansion.

However, even if the encoder is very careful and only includes effects that are guaranteed to hold after the expansion of a hierarchical operator, there are still potential problems. This is because a hierarchical operator might have several possible expansions, some of which result in some proposition holding, and others which result in the proposition not holding.

Hierarchical Promiscuity

A related, though slightly different problem, is one that Wilkins [Wilkins, 1988] terms *hierarchical promiscuity.* The problem occurs when operators are described abstractly, using different sets of predicates for each level of abstraction. It is possible then, when the planner expands different parts of the plan at different rates, that one part will be referring and modifying predicates at a much lower level than a part of the plan previous to it. In such situations it is possible that potentially harmful side effects at the lower level of abstractions will not be recognized until much later in the planning process, resulting in unnecessary planning.

There have been several solutions proposed for this problem. SIPE has a mechanism for enforcing an ABSTRIPS-like ordering in expanding the operators down to different abstraction levels. Further, it allows the specification of special delaying operators, which cause SIPE to refrain from planning for certain goals until some conditions have been satisfied in its state. Yang [Yang, 1989] proposes a solution wherein syntactic restrictions are computed for operators ahead of time which guarantee that harmful side-effects will not occur after expansion of an operator.

Unrecognized resolutions

However, even when the operators are specified completely accurately there are still potential problems. For example, take the problem in figure 7.4 proposed by Yang [Yang, 1989]. Part (a) represents a plan with two actions, each of which clobbers the other action's precondition. There is seemingly no legal ordering to the two action. However, when the plan is further expanded in (b) each action has become two actions and a legal ordering exists among the resulting four actions.

This is an instance of a "Double Cross" described by Sacerdoti [Sacerdoti, 1977]. In this situation a seemingly unresolvable conflict at one point in the plan can be resolved when the plan is further expanded. Thus, a planner using the traditional hierarchical operator specification might give up and backtrack when plan (a) is encountered, missing a potential solution.

Incompleteness

Another problem, less serious, but still interesting from a theoretical point of view is that of completeness. There has as yet not been a planner proposed which supports hierarchical operators and yet is complete. The reason this is not such a serious problem is that completeness in any planner implies intractability. Hence, any planner of practical value must make use of some sort of heuristic information to cut substantial portions of the search space. However, from a theoretical point of view, a completeness result provides a useful point of reference and starting point for discussing in

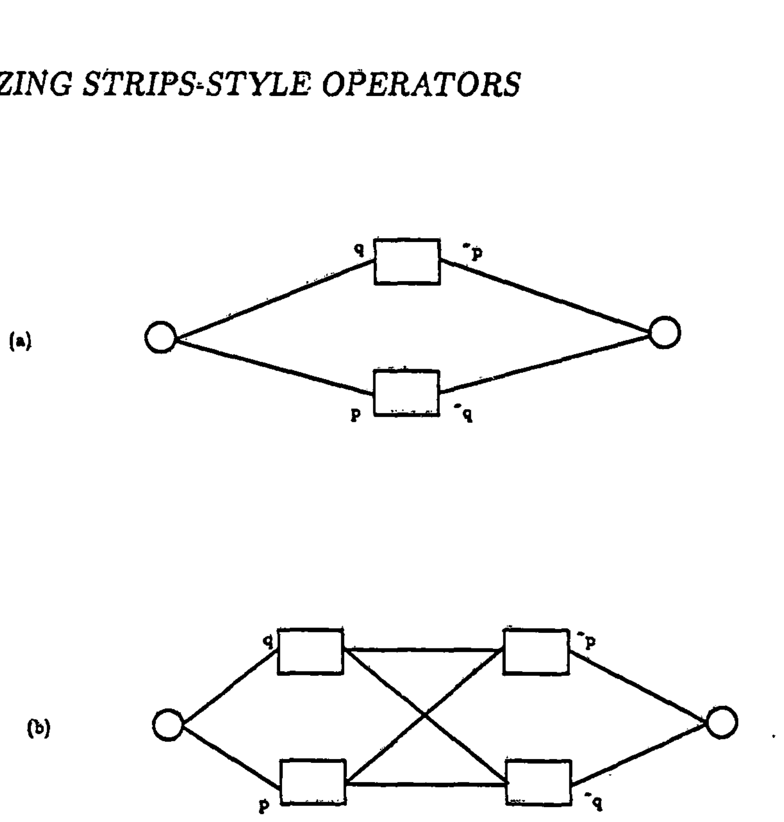


Figure 7.4: (a) A plan with seemingly unresolvable conflicts (b) Resolution of conflicts after reduction.

what ways a planner deviates from a complete algorithm.

7.2 Generalizing STRIPS-style operators

Although it allows for the formalization of restricted non-linear planning, the STRIPS-style operators used by TWEAK fail to capture one important aspect of most major non-linear planners, namely their hierarchical nature. It is no accident that NOAH, SIPE, etc. are generally referred to as *hierarchical* planners - this has traditionally been their defining characteristic and a source of much of their power.

In this chapter we present a generalization of the STRIPS-style operators that captures much of the hierarchical nature of previous planners. We then demonstrate a control strategy for reasoning with this generalized representation that guarantees a limited form of completeness.

7.3 Representation

We generalize STRIPS-style operators by defining hierarchical operators to be *distinguished* plans. An operator is simply a plan which has been deemed useful enough to store and use in problem solving. Obviously, STRIPS-style operators are special cases, namely they are plans limited to one action and possibly some codesignation constraints. More formally:

Definition 3 (Operator) *An operator is a triple $(\mathcal{O},\mathcal{T},\mathcal{C})$ where $\mathcal{O}$ is a set of actions, $\mathcal{T}$ is a set of temporal constraints, and $\mathcal{C}$ is a set of codesignation constraints.*

An operator is very much like a plan. A *primitive operator* is a specialization of operator.

Definition 4 (Primitive Operator) *A primitive operator is an operator $(\mathcal{O},\mathcal{T},\mathcal{C})$ where $\mathcal{O}$ is a unary set consisting of one action, $\mathcal{T}$ is an empty set, and $\mathcal{C}$ is a set of codesignation constraints.*

Before an operator $(\mathcal{O},\mathcal{T},\mathcal{C})$ can be used in a plan it must be instantiated. This is done by creating a new operator $(\mathcal{O}',\mathcal{T}',\mathcal{C}')$ where

- $\mathcal{O}'$ is a copy of $\mathcal{O}$ where every variable of every action of $\mathcal{O}$ is replaced with a new variable in the corresponding action of $\mathcal{O}'$.
- $\mathcal{T}'$ is a copy of $\mathcal{T}$ where every action in a temporal constraint is replaced by its corresponding copy.
- $\mathcal{C}'$ is a copy of $\mathcal{C}$ where every variable in the codesignation constraints is replaced by its corresponding copy.

After instantiating an operator we need to insert it into a plan. Given a plan $\mathcal{P}$ $(\mathcal{O}_p, \mathcal{T}_p, \mathcal{C}_p, \mathcal{S}_{initial}, \mathcal{S}_{final})$ and an instantiated operator $(\mathcal{O}_o, \mathcal{T}_o, \mathcal{C}_o)$ the new plan created by inserting the instantiated operator is $(\mathcal{O}_p \cup \mathcal{O}_o, \mathcal{T}_p \cup \mathcal{T}_o, \mathcal{C}_p \cup \mathcal{C}_o, \mathcal{S}_{initial}, \mathcal{S}_{final})$.

In figure 7.5 we give a graphical illustration of two hierarchical operators. In the figure, one of the hierarchial operators has been chosen to be inserted into the plan.

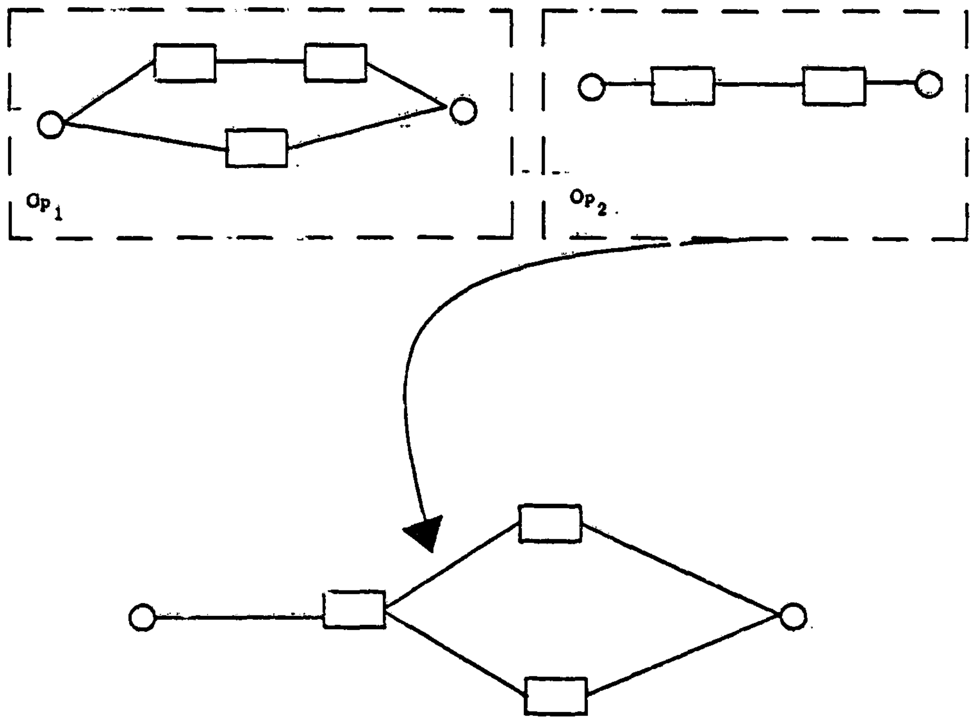


Figure 7.5: Diagram of Hierarchical Operators

This definition of hierarchical operators readily lends itself to the creation of new ones. The planner can create new operators simply by storing old plans which it deems to be potentially useful. Of course, only some plans will be particularly useful so the planner must have some means of deciding on the usefulness of particular hierarchical operators.

7.4 Hierarchical TWEAK

An important feature of our hierarchical operators is that at any point during planning, our plan is always composed solely of primitive actions. This feature allows us to use the TWEAK modal truth criterion to determine the necessary truth of goals and preconditions.

However, we need to extend TWEAK's control strategy to include hierarchical operators. As it turns out we only have to make a few minor changes to the algorithm in order to handle abstract operators.

7.4.1 Selecting Hierarchical Operators

One important issue we need to address is that of selecting hierarchical operators for instantiation into a plan. Previously, an operator was selected on the basis of whether any proposition in its add list could possibly codesignate with the proposition which needed to be achieved. Now that each operator is composed of a partial order of operators we need to decide what criteria to use when selecting an operator.

One solution is to choose only hierarchical operators for which the proposition of the current goal is possibly asserted in a hypothetical situation placed after all the primitive actions in the operator. Although this solution is intuitively appealing it is somewhat restrictive. Figure 7.6 illustrates a situation where we should have chosen a hierarchical operator even though, as a unit, it does not possibly assert the current goal.

In figure 7.6 if we need an action to achieve the precondition p of the plan in part (a), we can choose the hierarchical operator A and insert it into the plan as shown in part (b) of the figure. Note that we should choose operator A even though p is not possibly true after its application.

The solution we have chosen is to choose an operator for instantiation into a plan if *any* of its actions possibly asserts the current goal, even though one its later actions might deny it. Using this approach, Hierarchical TWEAK would have chosen to expand operator A because its subaction A_1 possibly asserts the goal proposition.

Hierarchical TWEAK is then simply TWEAK augmented with the hierarchical operator selection strategy outlined above, plus facilities for properly instantiating and inserting hierarchical operators.

7.5 Differences with Other Hierarchical Operators

Incomplete Specification of Operators

The problem of incorrectly specifying preconditions and effects of hierarchical operators is not an issue since our operators do not have explicit preconditions or effects

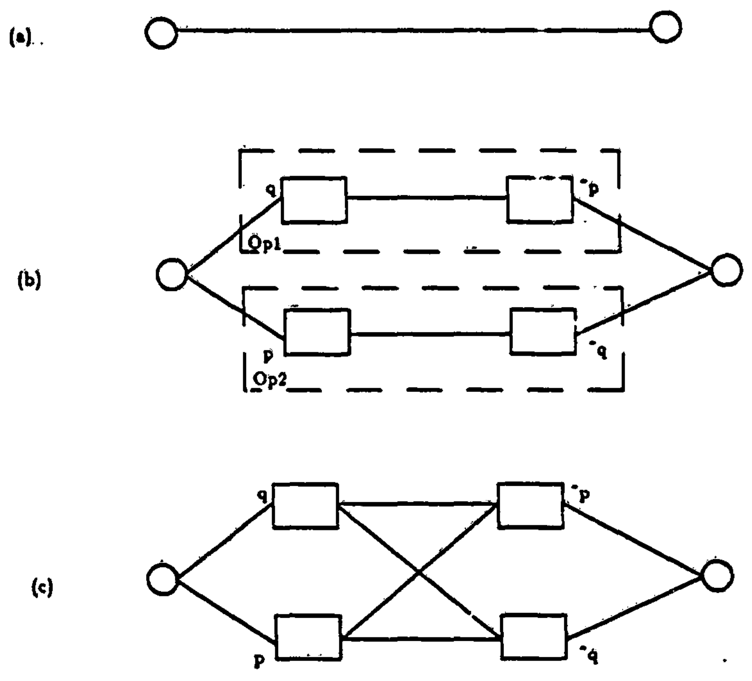


Figure 7.7: Yang's problem revisited

Completeness

An interesting question to consider is whether completeness is preserved with the new hierarchical operators. Clearly, if we retain as a condition that every primitive operator of the domain be represented by one hierarchical operator the new algorithm will remain complete, since any plan that would have been found without hierarchical operators will still be discovered.

However, if we relax this condition, we cannot guarantee completeness in the sense that if there exists a plan composed of primitive operators, one will be found using only hierarchical operators. One obvious counterexample is the case where the only final solution consists of exactly one primitive action, but every operator in the domain consists of at least two actions.

We can, though, guarantee a weaker form of completeness. Namely, if a plan that is a solution to a problem can be fully partitioned into sets of actions, each set being an instantiation of a hierarchical operator, Hierarchical TWEAK will discover it. We will refer to such a partition as a *hierarchical partition*.

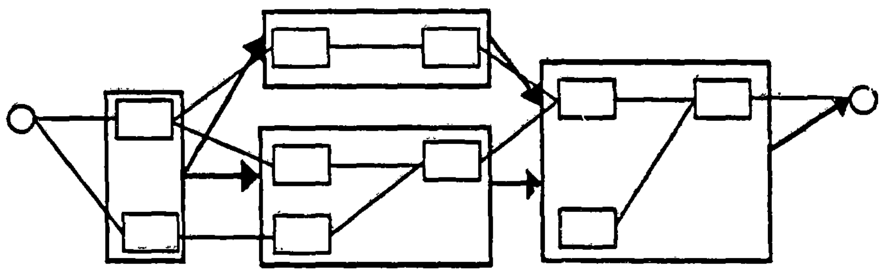


Figure 7.8: Partition Graph

This is most clearly explained in graph theoretic terms. We will think of the partitioned plan as a graph, where one partition points to another if the former contains a primitive action which establishes a proposition that is a precondition for an action of the latter. Further, a partition points to the final situation if some action in the partition establishes a proposition in the final situation. We will refer to this graph as *the partition graph*.

Definition 5 (Spanning Property) *A hierarchical partition of a plan satisfies the spanning property iff there is a path from every partition to the final situation.*

Lemma 7.5.1 *A hierarchical partition satisfying the spanning property has some partition that can be removed, such that the resulting plan still satisfies the spanning property.*

Proof (by contradiction):

Assume lemma 7.5.1 does not hold. Then it must be the case for every partition that removing it results in some partition no longer having a path to the final situation. This implies that for every partition p_1 there is some partition p_2 that points to it such that all the paths from p_2 to the final situation contain p_1. We will refer to partitions such as p_2 as *dependent* partitions.

Now, we start at the final situation. We choose any partition that establishes a proposition in the final situation. We mark it. We then traverse the partition graph, by choosing a partition that is dependent on the current one. We mark it and repeat the procedure. Note that we cannot revisit a marked partition since we already know there is a path from every marked partition to the final situation that does not contain any unmarked partitions. Therefore, a marked partition cannot be dependent on an unmarked one. Since the graph is finite, it must be the case that for some partition we will be unable to find another partition that is dependent on it. But this violates our assumption that such a partition exists for every partition. Therefore lemma 7.5.1 must hold. □

Lemma 7.5.2 *Every hierarchical partition of a plan generated by TWEAK satisfies the spanning property.*

Proof:

Define the temporal distance of a primitive action to be the longest path from the primitive action to the final situation over the temporal constraints in the plan.

Define the temporal distance of a partition to be the minimal temporal distance of its primitive actions. We prove that there must be a path from every partition to the final situation by induction on the temporal distance of partitions.

Base step: In the null plan there are no partitions and therefore the lemma holds trivially. In all other plans, if a partition's temporal distance is 1 it must be the case that one of its primitive actions establishes a proposition in the final situation (a primitive action must be necessarily before any situation in which it establishes a proposition), or it would not have been inserted by TWEAK. This means that the partition points to the final situation and therefore there is a path from the partition to the final situation.

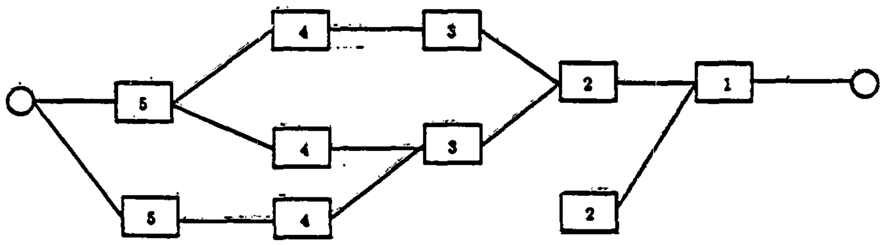


Figure 7.9: Temporal Distance

Induction step: Assume that for every partition with a temporal distance of $n-1$ or less there is a path in the partition graph to the final situation. We will prove that there is a path in the partition graph for all partitions with a temporal distance of n. In the partition there must be some primitive action such that its temporal distance is n. Furthermore, it must be the establisher of a proposition of either the final situation or the precondition of another primitive action. In the former case it is obvious that there is a path from the partition to the final situation. In the latter case it must be the case that the other primitive action is part of a different partition which must have a temporal distance of $n-1$ or less. But by the induction hypothesis there must exist a path from that partition to the final situation. Therefore, there is a path from the original partition to the final situation. □

We want to prove that every plan that is a solution to a problem and can be legally partitioned can be constructed by Hierarchical TWEAK. Since TWEAK is complete it can construct every such plan. But by lemma 7.5.2, every such plan must satisfy the spanning property. Therefore, it suffices to show:

Theorem 7.5.3 (Limited Completeness) *If a plan can be fully partitioned into n mutually exclusive sets of actions, each set being an instantiation of some hierarchical operator, such that the spanning property holds, the Hierarchical TWEAK algorithm will construct it.*

Proof (by induction on n, the number of partitions):

Base step: $n = 0$

If the plan can only be fully partitioned into 0 partitions then the plan must be the null plan. This is the plan that Hierarchical TWEAK begins with.

Induction step:

Assume the theorem holds for plans that can be partitioned into $n-1$ hierarchical operators such that the spanning property holds. We will show it must hold for all plans that can be partitioned into n hierarchical operators such that the spanning property holds.

By lemma 7.5.1 there must be some partition that can be removed from the plan such that the spanning property holds in the resulting partitioning. But because this is a plan of $n-1$ partitions and the spanning property holds, the induction hypothesis guarantees that hierarchical TWEAK will construct it. It now remains to be shown that the removed partition would be added. But since every partition is, by definition, an instantiation of a hierarchical operator, and since the spanning property holds, it must be the case that there is some primitive action of the partition that establishes a proposition of a situation outside the partition. Therefore, since Hierarchical TWEAK, in its complete breadth-first search, inserts all hierarchical operators which have some primitive action which can possibly establish some unachieved proposition in the plan, the hierarchical operator corresponding to the partition would also be inserted into the plan. Since the TWEAK declobbering strategy guarantees that all possible alternatives will be constructed, temporal and codesignation constraints would be added to the plan, such that one of the resulting plans was identical to the original plan. □

Therefore, if a solution exists to a problem, such that the resulting plan can be partitioned into n hierarchical operators, then Hierarchical TWEAK will find it.

Shuffling of Operators

Another, more subtle difference with the other hierarchical operator formalisms is that using the traditional hierarchical operators, once a hierarchical operator is inserted into a plan, its expansions must satisfy its higher level temporal constraints. For example, in figure 7.10 we have the expansion of a SIPE hierarchical operator.

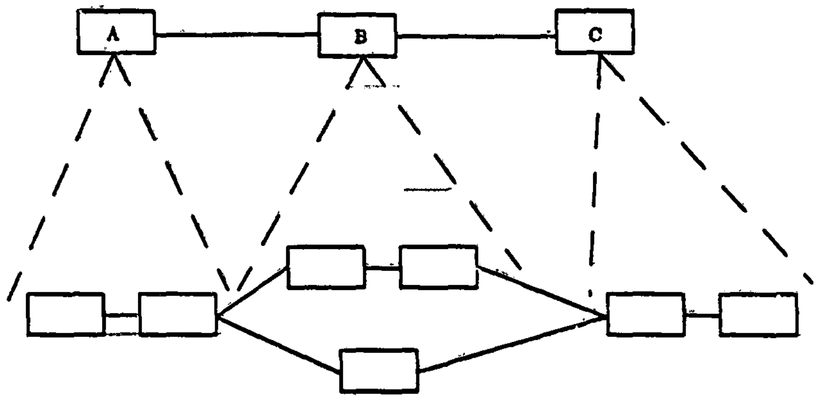


Figure 7.10: SIPE plan example

In this example, all the subactions of action B must be placed after all the subactions of action A and before all the subactions of action C. Using our definition of hierarchical operators it is possible that the plan could expand such that some of the subactions of an action could be shuffled with some of the subactions of another action. e.g. as we saw earlier, figure 7.6 provides an illustration of this. This feature provides Hierarchical TWEAK with more flexibility when expanding a plan.

7.6 PABLO Implementation

We have extended the PABLO operator representation to include hierarchical operators. We present an example that should help clarify the usefulness of hierarchical operators. We will use the robot domain used in STRIPS and ABSTRIPS. Before presenting an example problem we define two hierarchical operators.

Operator 1 allows the robot to get to an adjacent room even when the door is closed. Operator 2 is similar and allows the robot to push a box into another room

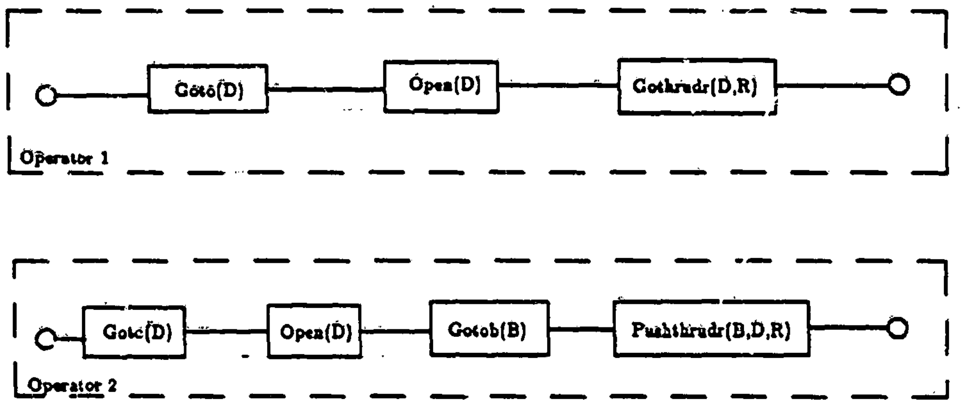


Figure 7.11: Hierarchical Operators for the Robot Domain

when the door is closed.

We will demonstrate PABLO on the problem depicted in figure 7.12.

See figure 7.13 for the final plan. PABLO solves the problem considerably faster when using the hierarchical operators than it does without.

7.7 Retaining useful plans

One useful consequence of our planner reasoning with hierarchical operators whose semantics are perspicuous to itself is that learning new hierarchical operators in considerably facilitated. In fact, to generate a new hierarchical operator the planner need only copy a current plan and store it along with its other operators. Of course, some criteria must be applied to decide which operators might be generally useful, and which might not.

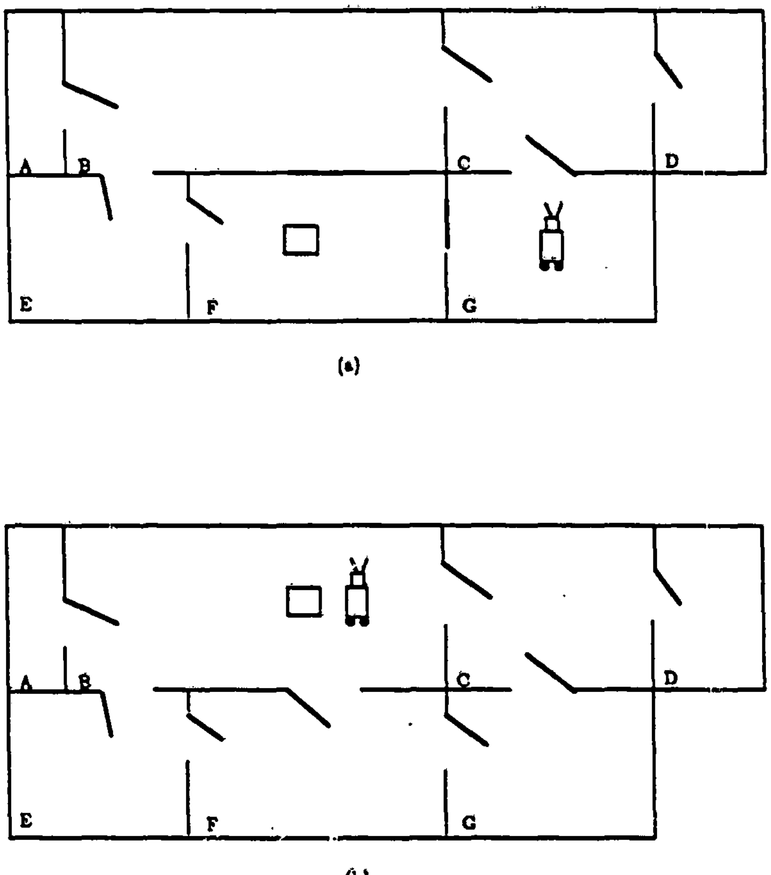


Figure 7.12: Robot Domain Problem. (a) Initial State, (b) Goal State.

7.7.1 Towers of Hanoi

How can hierarchical operators be learned and used effectively in planning? Here is one possible scenario in the Towers of Hanoi domain. The strategy of the planner is to solve progressively more difficult problems within the domain.

Suppose the planner is given the three disk Towers of Hanoi problem. First, it orders the subgoals independently in terms of difficulty. One way this could be done is by using the predicate relaxation definitions and applying them to the three goals. The resulting order would then be (1) Onpeg(A,P3), (2) Onpeg(B,P3), (3) Onpeg(C,P3).[1] The planner would then plan first for achieving Onpeg(A,P3). This

[1]Note the different notation from before, Onpeg(x,y) instead of On(x,y). This change was made to facilitate the exposition of this particular approach.

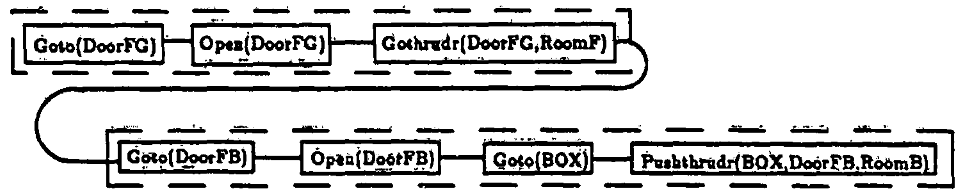


Figure 7.13: Solution using hierarchical operators.

is trivial and does not generate a new operator. It would then plan to achieve the conjunction of (1) and (2). This would be slightly more difficult and would generate the plan **Move(A,P2)-Move(B,P3)-Move(A,P3)**. This plan would be stored away for further use. But how should it be generalized? Clearly, one should not just convert all the constants in the plan into variables.

One possible generalization mechanism is to consider the bulk preconditions of the plan [Drummond and Currie, 1988]. The bulk preconditions are those that must hold in the state where the hierarchical operator will be applied to guarantee that every primitive action in the operator is applicable. If these preconditions have more than one consistent instantiation they should be generalized.

In the plan for solving goals (1) and (2) the bulk preconditions and their possible instantiations are as follows:

Clear(d1)	Clear(A)	Clear(A)
Onpeg(d1,p1)	Onpeg(A,P1)	Onpeg(A,P1)
Movable(d1)	Movable(A)	Movable(A)
Onpeg(d1,d2)	Onpeg(A,B)	Onpeg(A,B)
Onpeg(d2,p1)	Onpeg(B,P1)	Onpeg(B,P1)
Movable(d2)	Movable(B)	Movable(B)
Onpeg(d3,p2)	Onpeg(BASE2,P2)	Onpeg(BASE3,P3)
Smaller(d1,d3)	Smaller(A,BASE2)	Smaller(A,BASE3)
Clear(d3)	Clear(BASE2)	Clear(BASE3)
Onpeg(d4,p3)	Onpeg(BASE3,P3)	Onpeg(BASE2,P2)
Smaller(d2,d4)	Smaller(B,BASE3)	Smaller(B,BASE2)
Clear(d4)	Clear(BASE3)	Clear(BASE2)

There are two possible instantiations of the bulk preconditions in the initial state. In these instantiations p2, p3, d3, and d4 are instantiated to different values. The variable p2 is either instantiated to P2 or P3, p3 is either instantiated to P3 or P2, d3 is either instantiated to BASE2 or BASE3, and d4 is either instantiated to BASE3 or BASE2. The fact that only these variables vary suggests that only these variables should be generalized.

After generalizing them the plan for solving goals (1) and (2) becomes **Move(A,x) - Move(B,y) - Move(A,y)**. Finally, the planner proceeds to solve goals (1), (2), and (3). The actual solution trace can be seen in figure 7.14. The planner uses the newly generated hierarchical operator for moving the two top blocks twice, once to move them to the middle peg and finally to move them to the last peg.

The key to this approach is first to order the goals in terms of difficulty. Predicate relaxation provides a mechanism for doing so. Secondly, using the initial situation to determine the possible instantiations of the bulk preconditions of the generated hierarchical operator, to decide which variables should be generalized. We believe this to be a promising approach to automatic operator abstraction, but more work remains to be done in this area.

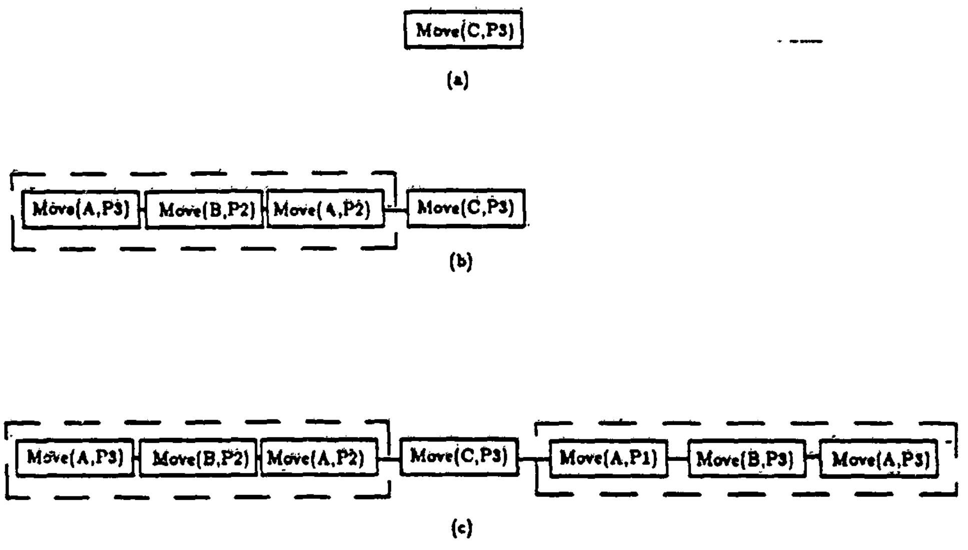


Figure 7.4: Operator Abstraction Solution Trace

7.8 Conclusion

We have presented an elegant definition of hierarchical operators which overcomes many of the problems associated with earlier hierarchical operator definitions, and proven that it is possible to guarantee limited completeness when using them in planning. A mechanism for using hierarchical operators has been incorporated into PABLO.

Chapter 8

Combining Abstraction Methods

8.1 Introduction

In this chapter we demonstrate several ways to use state abstraction and operator hierarchicalization simultaneously for effective problem solving. Recall that PABLO achieves a form of state abstraction through predicate relaxation and operator hierarchicalization through the use of hierarchical operators.

8.2 Robot World Example

In this first example we will use predicate relaxation and hierarchical operators in the manner in which they have been presented. We will see later how predicate relaxation can be extended to include relaxation of predicates over hierarchical operators.

We will present an example in some detail and describe the problem solving that PABLO does to solve it. The domain of the example is the familiar robot world, with rooms, doors, boxes. In addition to these we also include keys and add the operators to lock and unlock doors. Furthermore, keys can be dropped into boxes, in which case they can no longer be retrieved by the robot.

In the following operator descriptions some arguments to predicates in some of the delete_lists are preceded by a \$. These variables are special in the sense that they are treated as global variables. For example, if there is a predicate $P(\$1)$ in a delete list, and two predicates $P(A)$ and $P(B)$ in the situation description, then both predicates will be deleted from the situation description, instead of just one. These are the operators used:

Pickup-key(R,k)

P:{Type(k,Key),Nextto(R,k),Graspable(k)}

D:{Nextto(R,k)}

A:{Holding(R,k)}

Put-key-in-box(R,k,b)

P:{Type(k,Key),Type(b,Box),Nextto(R,b),Holding(R,k)}

D:{Holding(R,k),Graspable(k)}

A:{In(k,b)}

Goto-object(R,o)

P:{Type(o,Object),Inroom(R,rx),Inroom(o,rx)}

D:{Nextto(R,$1)}

A:{Nextto(R,o)}

Goto-door(R,d)

P:{Type(d,Door),Inroom(R,rx),Connects(d,rx,ry)}

D:{Nextto(R,$1)}

A:{Nextto(R,d)}

Gothru-door(R,d)

P:{Type(d,Door),Inroom(R,rx),Connects(d,rx,ry),Status(d,Open)}

D:{Nextto(R,$1),Inroom(R,rx)}

A:{Inroom(R,ry)}

Open-door(R,d)

P:{Type(d,Door),Nextto(R,d),Status(d,Closed)}

D:{Status(d,Closed)}

A:{Status(d,Open)}

Close-door(R,d)

P:{Type(d,Door),Nextto(R,d),Status(d,Open)}

D:{Status(d,Open)}

A:{Status(d,Closed)}

Lock-door(R,d,k)

P:{Type(d,Door),Nextto(R,d),Status(d,Closed),Holding(R,k),Type(k,Key)}

D:{Status(d,Closed)}

A:{Status(d,Locked)}

Unlock-door(R,d,k)

P:{Type(d,Door),Nextto(R,d),Status(d,Locked),Holding(R,k),Type(k,Key)}

D:{Status(d,Locked)}

A:{Status(d,Closed)}

In addition we have defined the following hierarchical operators:

Goto-and-Pickup-key(R,k)

Goto-object(R,k), Pickup-key(R,k)

Goto-and-Put-key-in-box(R,k,b)

Goto-object(R,b), Put-key-in-box(R,k,b)

Goto-and-Lock-door(R,k,d)

Goto-door(R,d), Lock-door(R,k,d)

Goto-and-Unlock-door(R,k,d)

Goto-door(R,d), Unlock-door(R,k,d)

Goto-and-Close-door(R,d)

Goto-door(R,d), Close-door(R,d)

Goto-and-Open-door(R,d)

Goto-door(R,d), Open-door(R,d)

Pickup-and-Put-key-in-box(R,k,b)

Goto-object(R,k), Pickup(R,k), Goto-object(R,b), Put-key-in-box(R,k,b)

Pickup-and-Lock-door(R,k,d)

Goto-object(R,k), Pickup(R,k), Goto-door(R,d), Lock-door(R,k,d)

Each of the above operators is a linear sequence of primitive operators. The codesignation constraints between their arguments is made explicit by substituting the same variable name for codesignating arguments.

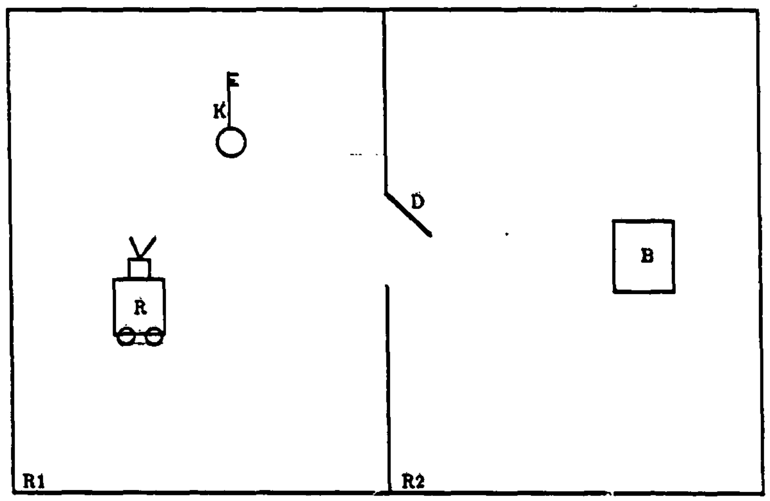


Figure 8.1: Robot World Problem

We will solve the problem in figure 8.1 The goal of the problem is to achieve $Status(D, Locked)$ and $In(K, B)$. This problem is an example of ***very strong interaction***. This type of interaction is more serious than the ***strong interaction*** encountered in Sussman's Anomaly [Sussman, 1973]. The difference is that in Sussman's Anomaly, once a plan has been developed to achieve each goal independently, it is possible to correct the plan simply by adding new actions; in this case the action of putting block C on the table. However, in the robot example we have presented it is not possible to repair the plan in this manner, just given the two plans for achieving each goal.

8.2.1 One Level of Abstraction

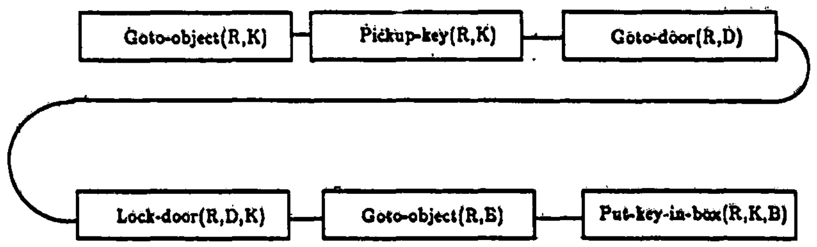


Figure 8.2: Plan at first level of abstraction

We first solve the problem using only one level of predicate relaxation abstraction. The resulting plan at the first level of abstraction can be seen in figure 8.2. PABLO has used two hierarchical operators, **Pickup-and-Put-key-in-box** and **Goto-and-Lock-door**. Furthermore, it has interleaved the primitive actions of the two operators. This is something most other hierarchical operator formalisms do not allow. There are two predicates in this plan that are abstractly satisfied. They are $\text{Status}^1_{\text{rel}}$(door,Closed) and $\text{Inroom}^1_{\text{rel}}$(Robot,R2).

In figure 8.3 we have the result of planning at the base level of abstraction. Notice that both abstract predicates have been satisfactorily achieved. Furthermore, the optimal plan is produced. It should be pointed out that problems with very strong interaction pose difficulties for many planners. ABSTRIPS would not be able to solve the above problem.

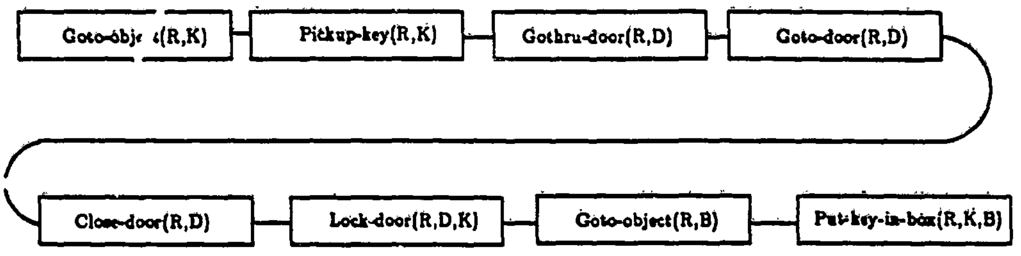


Figure 8.3: Plan at base level

8.2.2 Two Levels of Abstraction

Instead of just relaxing one level of abstraction we can relax the predicates two levels. Doing so results in the plan seen in figure 8.4. The relaxed predicates that hold are $\text{Inroom}^2_{\text{rel}}$(Robot,R2), $\text{Holding}^2_{\text{rel}}$(k), $\text{Status}^2_{\text{rel}}$(door,Locked).

Continued planning at the first level of abstraction and at the base level results in the same plans as in our previous example, albeit achieved using different hierarchical operators. PABLO arrived at the correct plan in two different ways depending on the amount the predicates were relaxed.

8.3 Generalizing Predicate Relaxation

It is also possible to generalize predicate relaxation so that predicates are regressed over hierarchical operators. To determine the desired regressed expression, we build a hypothetical abstract operator whose add list is the union of the add lists of the

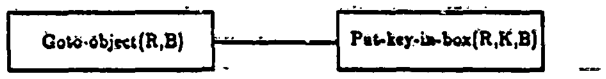


Figure 8.4: Plan at the second abstraction level

primitive actions of the hierarchical operator. The preconditions of the operator are the *bulk preconditions* [Drummond and Currie, 1988] of the hierarchical operator. The bulk preconditions are those that must hold in the state where the hierarchical operator will be applied to guarantee that every primitive action in the operator is applicable. Therefore, the precondition of the hierarchical operator becomes the conjunction of the preconditions of the primitive actions that are not necessarily true. For our purposes the delete list is not important, since any expression resulting from the regression that contains a proposition from the delete list is subsumed by the proposition itself.

Having created this operator, it can be used as a primitive action, for the purpose of regressing predicates through it.

8.3.1 Shift of Semantics

Now that we have modified predicate relaxation it is important to discuss the implications. Before, if a relaxed predicate held at level n we were guaranteed that there existed a plan of n actions or less that achieved that predicate. Now, we are guaranteed that there exists a plan of n hierarchical operators or less. But this is reasonable in light of the fact that the predicate relaxation is a measure of the difficulty of planning to achieve a particular predicate. The number of hierarchical operators necessary to achieve a predicate is a good measure of this difficulty.

8.4 ABSTRIPS domain

We will demonstrate the relaxation of predicates over hierarchical operators in the ABSTRIPS domain. The operators are those described in [Sacerdoti, 1974] which are essentially the same as those described in [Fikes and Nilsson, 1971], with the exception of two which are not used in the following examples.

Gotob(R,b)

P:{Type(b,Box),Inroom(R,rx),Inroom(b,rx)}

D:{Nextto(R,$1)}

A:{Nextto(R,b)}

Goto(R,d)

P:{Type(d,Door),Inroom(R,rx),Connects(d,rx,ry)}

D:{Nextto(R,$1)}

A:{Nextto(R,d)}

Pushb(R,bx,by)

P: {Type(by,Object),Pushable(bx),Nextto(R,bx),Inroom(bx,rx), Inroom(by,rx),Inroom(R,rx)

D:{Nextto(R,$1),Nextto(bx,$2),Nextto($2,bx)}

A:{Nextto(bx,by),Nextto(by,bx),Nextto(R,bx)}

Pushd(R,dx,bx)

P:{Type(dx,Door),Pushable(bx),Nextto(R,bx),Inroom(bx,rx), Connects(dx,rx,ry),Inroom(R,rx)

D:{Nextto(R,$1),Nextto(bx,$2),Nextto($2,bx)}

A:{Nextto(bx,dx),Nextto(R,bx)}

Gothrudr(R,d,ry)

P:{Type(d,Door),Inroom(R,rx),Connects(d,rx,ry),Status(d,Open)}

D:{Nextto(R,$1),Inroom(R,rx)}

A:{Inroom(R,ry)}

Pushthrudr(R,bx,dx,rx)

P:{Pushable(bx),Type(dx,Door),Type(rx,Room),Nextto(bx,dx),

Nextto(R,bx),Inroom(bx,ry),Inroom(R,ry),Connects(dx,ry,rx),Status(dx,Open)}
D:{Nextto(R,$1),Nextto(bx,$1),Nextto($1,bx),Inroom(R,ry),Inroom(bx,ry)}
A:{Inroom(bx,rx),Inroom(R,rx),Nextto(R,bx)}

Open(R,d)
P:{Type(d,Door),Nextto(R,d),Status(d,Closed)}
D:{Status(d,Closed)}
A:{Status(d,Open)}

Close(R,d)
P:{Type(d,Door),Nextto(R,d),Status(d,Open)}
D:{Status(d,Open)}
A:{Status(d,Closed)}

We have also defined the following hierarchical operators:

Gothruclosedd r(R,dx,ry)
Goto(R,dx), Open(dx), Gothrudr(R,dx,ry)

Gotob-and-Pushb(R,bx,by)
Goto(R,bx), Pushb(R,bx,by)

8.4.1 Managing the Size of Relaxation Expressions

As the number of operators grows in a domain it is important to consider ways to limit the size of the relaxation expressions. There are several methods PABLO uses to limit these sizes.

Removing Subsumed Disjuncts

The relaxation definitions are kept in disjunctive normal form. During the relaxation it often happens that one disjunct subsumes another one. In such situations PABLO removes the subsumed disjunct. There is no reason for retaining it, since whenever it holds, the disjunct which subsumes it will also hold. See chapter 6 for an example where this is done for the Clear(x) predicate in the blocks world. Once hierarchical operators are introduced the frequency of subsumed expressions naturally rises and this operation can lead to substantial savings.

Using Domain Knowledge

Another useful method to limit the size of the relaxed expressions is to use domain axioms to collapse disjuncts. For example, in the Robot World domain without locked doors we have the axiom Status(x,Closed) $\Rightarrow \neg$ Status(x,Open). The result of relaxing the Status predicate one level is the following:

$\text{Status}^1_{\text{rel}}(x,y)$	
Status(x,y)	[]
Nextto(Robot,x) $\wedge$ Type(x,Door) $\wedge$ Status(x,Open)	[Close(x)]
Nextto(Robot,x) $\wedge$ Type(x,Door) $\wedge$ Status(x,Closed)	[Open(x)]

However, using the domain axiom, the above can be collapsed to:

$\text{Status}^1_{\text{rel}}(x,y)$	
Status(x,y)	[]
Nextto(Robot,x) $\wedge$ Type(x,Door)	[Close(x) $\vee$ Open(x)]

This technique can lead to considerable simplification in the relaxed expressions.

Example in the ABSTRIPS Domain

To demonstrate how these techniques can lead to substantial savings we show the result of applying them to the relaxation expression of the Inroom(x,y) predicate. Without any simplification the result of relaxing Inroom(x,y) two levels, over the ABSTRIPS operators, can be seen in tables 8.1 and 8.2.

Obviously this expression is unacceptably large. However, if we make use of the techniques for managing the size of relaxed expressions the resulting expression, in table 8.3 is considerably more compact.

Inroom$^2_{rel}(x,y)$
Inroom(x,y) ∨
connects(z,r,y) ∧ type(z,door) ∧ status(z,closed) ∧ type(y,room) ∧ type(f,door) ∧
type(r,room) ∧ connects(f,g,r) ∧ status(f,open) ∧ inroom(robot,g) ∨
connects(z,r,y) ∧ type(z,door) ∧ status(z,closed) ∧ type(y,room) ∧ inroom(robot,g) ∧
connects(f,g,r) ∧ type(f,door) ∧ status(f,closed) ∧ type(r,room) ∨
type(v,door) ∧ type(y,room) ∧ connects(v,w,y) ∧ status(v,open) ∧ type(f,door) ∧
type(w,room) ∧ connects(f,g,w) ∧ status(f,open) ∧ inroom(robot,g) ∨
type(v,door) ∧ type(y,room) ∧ connects(v,w,y) ∧ inroom(robot,w) ∧ type(v,door) ∧
status(v,closed) ∧ nextto(robot,v) ∨
type(v,door) ∧ type(y,room) ∧ connects(v,w,y) ∧ status(v,open) ∧ inroom(robot,g) ∧
connects(f,g,w) ∧ type(f,door) ∧ status(f,closed) ∧ type(w,room) ∨
pushable(x) ∧ nextto(x,k) ∧ nextto(robot,x) ∧ type(k,door) ∧ type(y,room) ∧ sta-
tus(k,open) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(f,door) ∧ type(s,room) ∧ con-
nects(f,g,s) ∧ status(f,open) ∧ inroom(robot,g) ∨
pushable(x) ∧ nextto(x,k) ∧ nextto(robot,x) ∧ type(k,door) ∧ type(y,room) ∧ in-
room(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(k,door) ∧ status(k,closed) ∧
nextto(robot,k) ∨
pushable(x) ∧ nextto(robot,x) ∧ type(k,door) ∧ type(y,room) ∧ status(k,open) ∧
inroom(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(k,door) ∧ pushable(x) ∧
nextto(robot,x) ∧ inroom(x,g) ∧ connects(k,g,h) ∧ inroom(robot,g) ∨
pushable(x) ∧ nextto(x,k) ∧ type(k,door) ∧ type(y,room) ∧ status(k,open) ∧ in-
room(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(x,door) ∧ pushable(robot) ∧
nextto(robot,robot) ∧ inroom(robot,g) ∧ connects(x,g,h) ∧ inroom(robot,g) ∨
pushable(x) ∧ nextto(robot,x) ∧ type(k,door) ∧ type(y,room) ∧ status(k,open) ∧
inroom(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(k,object) ∧ pushable(x) ∧
nextto(robot,x) ∧ inroom(x,g) ∧ inroom(k,g) ∧ inroom(robot,g) ∨
pushable(x) ∧ nextto(robot,x) ∧ type(k,door) ∧ type(y,room) ∧ status(k,open) ∧
inroom(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(x,object) ∧ pushable(k) ∧
nextto(robot,k) ∧ inroom(k,g) ∧ inroom(x,g) ∧ inroom(robot,g) ∨
pushable(x) ∧ nextto(x,k) ∧ type(k,door) ∧ type(y,room) ∧ status(k,open) ∧ in-
room(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(x,object) ∧ pushable(robot) ∧
nextto(robot,robot) ∧ inroom(robot,g) ∧ inroom(x,g) ∧ inroom(robot,g) ∨

Table 8.1: First half of relaxation expression.

pushable(x) $\wedge$ nextto(x,k) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ type(robot,object) $\wedge$ pushable(x) $\wedge$
nextto(robot,x) $\wedge$ inroom(x,g) $\wedge$ inroom(robot,g) $\wedge$ inroom(robot,g) $\vee$
pushable(x) $\wedge$ nextto(robot,x) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ type(k,box) $\wedge$ inroom(k,g) $\wedge$ in-
room(robot,g) $\vee$
pushable(x) $\wedge$ nextto(x,k) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ type(x,box) $\wedge$ inroom(x,g) $\wedge$ in-
room(robot,g) $\vee$
pushable(x) $\wedge$ nextto(robot,x) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ type(k,door) $\wedge$ connects(k,g,h) $\wedge$ in-
room(robot,g) $\vee$
pushable(x) $\wedge$ nextto(x,k) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ type(x,door) $\wedge$ connects(x,g,h) $\wedge$ in-
room(robot,g) $\vee$
pushable(x) $\wedge$ nextto(robot,x) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ inroom(robot,g) $\wedge$ inroom(x,g) $\wedge$
type(x,box) $\wedge$ inroom(k,h) $\wedge$ pushable(x) $\wedge$ type(k,object) $\vee$
pushable(x) $\wedge$ nextto(robot,x) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ inroom(robot,g) $\wedge$ inroom(k,g) $\wedge$
type(k,box) $\wedge$ inroom(x,h) $\wedge$ pushable(k) $\wedge$ type(x,object) $\vee$
pushable(x) $\wedge$ nextto(x,k) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ inroom(robot,g) $\wedge$ type(robot,box) $\wedge$
inroom(x,h) $\wedge$ pushable(robot) $\wedge$ type(x,object) $\vee$
pushable(x) $\wedge$ nextto(x,k) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ status(k,open) $\wedge$ in-
room(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ inroom(robot,g) $\wedge$ inroom(x,g) $\wedge$
type(x,box) $\wedge$ pushable(x) $\wedge$ type(robot,object) $\vee$
pushable(x) $\wedge$ nextto(x,k) $\wedge$ nextto(robot,x) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ sta-
tus(k,open) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\wedge$ inroom(robot,g) $\wedge$ connects(f,g,s) $\wedge$
type(f,door) $\wedge$ status(f,closed) $\wedge$ type(s,room) $\vee$
pushable(x) $\wedge$ nextto(x,k) $\wedge$ nextto(robot,x) $\wedge$ type(k,door) $\wedge$ type(y,room) $\wedge$ sta-
tus(k,open) $\wedge$ inroom(robot,s) $\wedge$ inroom(x,s) $\wedge$ connects(k,s,y) $\vee$
type(v,door) $\wedge$ type(y,room) $\wedge$ connects(v,w,y) $\wedge$ status(v,open) $\wedge$ inroom(robot,w) $\vee$
inroom(robot,r) $\wedge$ connects(z,r,y) $\wedge$ type(z,door) $\wedge$ status(z,closed) $\wedge$ type(y,room)

Table 8.2: Second half of relaxation expression.

$\text{Inroom}^2_{\text{rel}}(x,y)$
connects(z,r,y) ∧ type(z,door) ∧ type(y,room) ∧ inroom(robot,g) ∧ connects(f,g,r) ∧ type(f,door) ∧ type(r,room) ∨
pushable(x) ∧ nextto(x,k) ∧ nextto(robot,x) ∧ type(k,door) ∧ type(y,room) ∧ inroom(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ nextto(robot,k) ∨
pushable(x) ∧ nextto(x,k) ∧ type(k,door) ∧ type(y,room) ∧ status(k,open) ∧ inroom(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y) ∧ type(x,box) ∨
inroom(robot,r) ∧ connects(z,r,y) ∧ type(z,door) ∧ type(y,room) ∨
pushable(x) ∧ nextto(x,k) ∧ nextto(robot,x) ∧ type(k,door) ∧ type(y,room) ∧ status(k,open) ∧ inroom(robot,s) ∧ inroom(x,s) ∧ connects(k,s,y)

Table 8.3: Relaxed expression using the simplification filters.

While simplifying the expression we made use of the domain constraint

$$\text{Status}(x,\text{closed}) \Rightarrow \neg\text{Status}(x,\text{open})$$

Notice that by using this constraint PABLO can capture the notion that it does not matter whether a door is open or closed, since PABLO has operators for either case (Gothrudr and Gothrucloseddr). All that is important is that there is a door between two rooms.

We can see that with a few simple techniques it is possible to achieve a sizable reduction in the size of the relaxed expressions. If the expression becomes unmanageably large even using these simplification techniques, PABLO stops relaxing the predicate. The user can set a threshold for the maximum allowable size for each relaxation predicate. This might result in PABLO performing additional planning at a higher level than it otherwise would, but this might be preferable to having an enormous and unwieldy relaxation expression to evaluate.

8.4.2 Example from ABSTRIPS

We gave PABLO the problem presented in [Sacerdoti, 1974] for purposes of comparison. The problem can be seen in figure 8.5. In this figure (a) represents the initial state and (b) is the goal state.

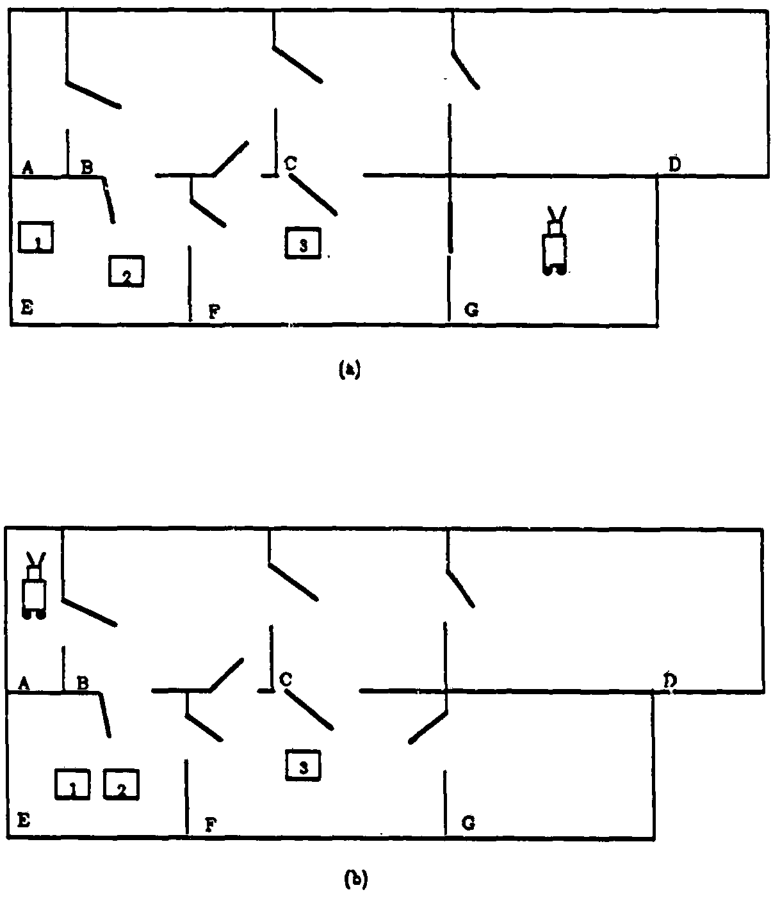


Figure 8.5: Problem that ABSTRIPS solves

PABLO begins planning at abstraction level 3. The plans generated after each abstraction level can be seen in figure 8.6. ABSTRIPS also uses four levels of abstractions for this problem.

The plans generated at each abstraction level by PABLO and ABSTRIPS are remarkably similar. This is probably due mostly to the nature of the problem and domain. ABSTRIPS is a linear planner and this is a linear problem, without strongly interacting subgoals. It is not surprising that both planners would generate similar plans at different levels of abstraction.

Given a different problem in a similar domain, such as the previous problem with

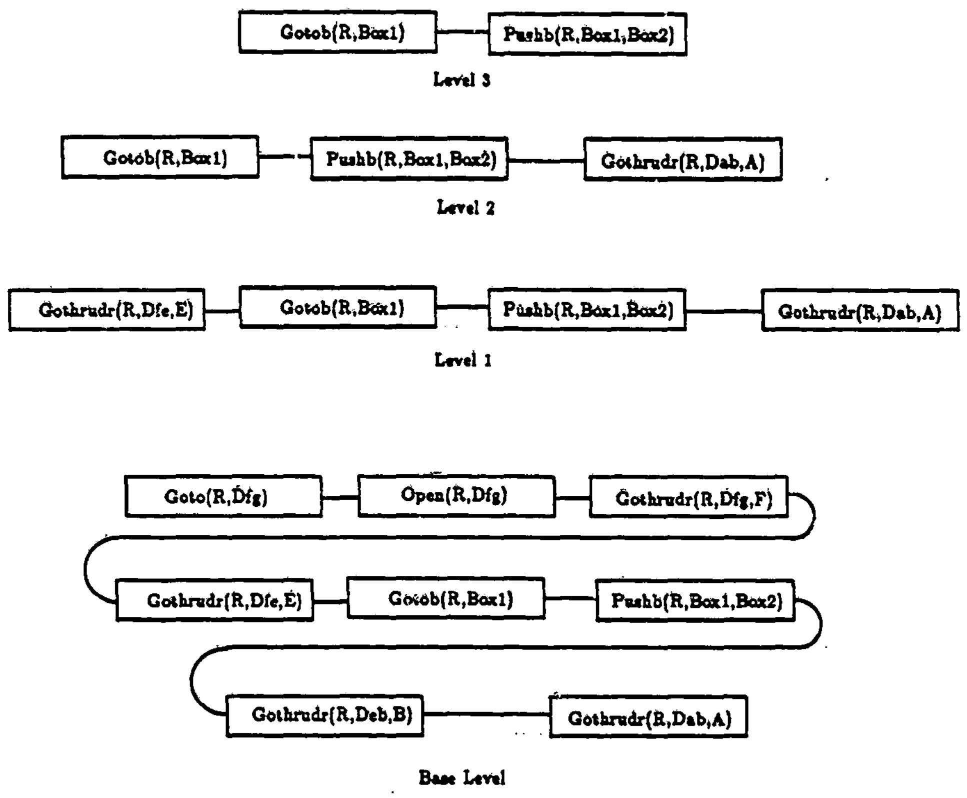


Figure 8.6: PABLO's solution

very strongly interacting goals the similarity obviously ends, since ABSTRIPS is unable to solve the problem. Also, given a different domain, such as the Towers of Hanoi, the two planners' behaviour might differ considerably. In the Towers of Hanoi domain, ABSTRIPS can use only one level of abstraction, no matter the complexity of the problem given, whereas PABLO generates $n - 1$ levels of abstraction for n disks.

8.5 Summary

We have presented two examples where state and operator abstraction are combined to produce interesting planning behaviour. We have extended predicate relaxation to include the relaxation of predicates over hierarchical operators. Finally, we demonstrated techniques whereby the size of predicate relaxation expressions can be substantially reduced.

Chapter 9

Classical Truth Criterion

9.1 Introduction

A truth criterion defines the conditions under which a predicate is true in a particular situation of a plan. Such a criterion is important since a planner must often check the truth of propositions during planning, e.g. to determine if a proposition of a precondition is satisfied. Because the underlying plan representation varies from planner to planner the truth criteria of planners have varied as well.

As we shall see, in some special plans, namely those where the actions are linearly ordered and where every predicate is ground, defining a truth criterion is relatively straightforward. Once we introduce variables, nonlinearity, conditional actions, deductive rules, typed variables, etc., defining a truth criterion becomes more complicated. For example, once we introduce nonlinearity, the truth of a predicate at a certain point in the plan depends on the possible orders of the actions preceding the point of interest. In this case we no longer speak simply about the truth of a proposition, but about the possible or necessary truth of a proposition.

The first formal definition of a truth criterion for partially ordered plans can be found in [Chapman, 1987]. Chapman refers to this criterion as the *Modal Truth Criterion.* The Modal Truth Criterion was defined for a particular plan formalism: Chapman's TWEAK formalism. As Chapman points out, TWEAK is a very restricted planning formalism. Further, even minor extensions to the formalism results in the truth criterion no longer being valid.

In this chapter, we discuss the previous work by Chapman, and then present a new planning ontology that is powerful enough to capture most planning formalisms proposed up until now. We then present a new *Classical Truth Criterion* for this planning formalism. This truth criterion is proved sound and complete.[1] Finally, we discuss some of the implications of the Classical Truth Criterion, and present an algorithm for checking the truth of a predicate in a plan.

9.2 Modal Truth Criterion

Chapman introduces the following truth criterion:

Definition 6 (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).*

In Chapman's logical notation. the criterion reads as follows:

$$
\begin{array}{l}
\exists t\ \Box t \preceq s \wedge \Box \mathit{assertedin}(p,t) \wedge \\
\quad \forall C\ \Box s \preceq C \vee \\
\qquad \forall q\ \Box \neg \mathit{denies}(C,q) \vee \\
\qquad\quad \Box q \not\approx p \vee \\
\qquad\quad \exists W\ \Box C \prec W \wedge \\
\qquad\qquad \Box W \prec s \wedge \\
\qquad\qquad \exists r\ \mathit{asserts}(W,r) \wedge p \approx q \Rightarrow p \approx r
\end{array}
$$

[1]A truth criterion is sound if whenever it holds for a predicate p and a situation s the predicate p is true in situation s. A truth criterion is complete if whenever p is true in situation s the truth criterion holds for predicate p in situation s.

There is a typo in Chapman's logical formulation of the Modal Truth Criterion. In order to make the formula conform to the English version we need to replace

$\exists r\ asserts(w,r) \wedge p \approx q \Rightarrow p \approx r.$

with

$\exists r\ asserts(w,r) \wedge \Box(p \approx q \Rightarrow p \approx r).$

In what follows we will refer to this criterion as *MTC*. The MTC is proven by Chapman to be valid for plans represented in TWEAK's formalism. For a complete description of TWEAK see [Chapman, 1987]. An action in TWEAK has a precondition and a postcondition, each of which are sets of predicates which must hold before the application of the action and after the application of the action respectively. Plans in TWEAK are partial orders of actions. The TWEAK formalism explicitly excludes restricting variables to a finite domain, conditional actions, and deductive rules. Chapman notes that if TWEAK is extended in any of these ways the Modal Truth Criterion fails. For example, in order to guarantee,

$\neg MTC \Rightarrow \neg\Box Holds(p,s)$

or

$$
\begin{aligned}
&\forall t\ \Diamond s \prec t \vee \Diamond\neg asserts(p,t) \vee \\
&\quad \exists c\ \Diamond c \preceq s \wedge \\
&\qquad \exists q\ \Diamond denies(c,q) \wedge \\
&\qquad\quad \Diamond q \approx p \wedge \\
&\qquad\quad \forall w\ \Diamond w \preceq c \vee \\
&\qquad\qquad \Diamond s \preceq w \vee \\
&\qquad\qquad \forall r\ \neg asserts(w,r) \vee \Diamond(p \approx q \wedge p \not\approx r) \\
&\Rightarrow \neg\Box Holds(p,s)
\end{aligned}
$$

it must be the case that if no action necessarily asserts p then p cannot necessarily hold. Although this is the case for the plan representation used in TWEAK it is not the case for many representations which are just somewhat more expressive. For example, if we allow restricted ranges on variables we can have situations where no step necessarily asserts a proposition but where the proposition is asserted by some step in all ground linear completions of the plan. Take the plan in figure 9.1.

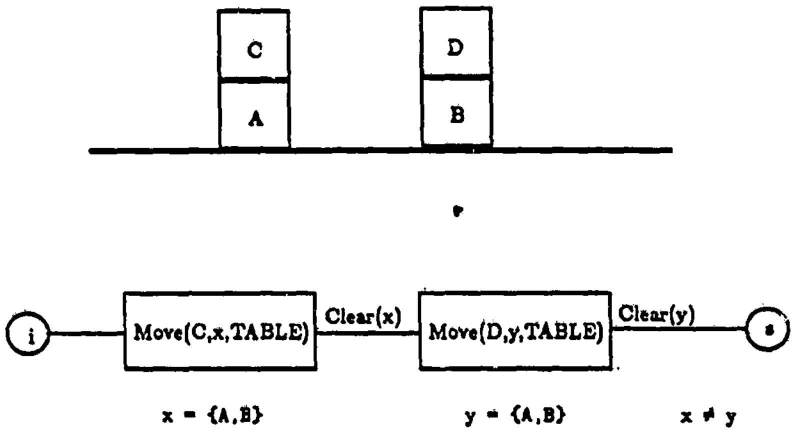


Figure 9.1: Restricted Range Plan.

In this plan Clear(A) holds in situation s, however the Modal Truth Criterion fails to determine this.

Restricting the range of variables can greatly reduce the planning time in certain cases. SIPE is the main planner which makes use of this feature for great computational savings. A truth criterion that can deal with this extension is therefore an important contribution.

We have a similar problem if we extend our plan representation to allow arbitrary deduction performed in situations. We have to be careful when extending our language to handle this. Lifschitz [Lifschitz, 1986] has shown that for a planner which uses STRIPS style operators to remain sound it is necessary that any axioms we use must hold in all states of the domain. However, if we allow such axioms in our plan language the completeness proof of the Modal Truth Criterion fails since we can now have two actions which synergistically assert a proposition in a way similar to that of figure 9.1.

For example, suppose we are in a blocks world which allows more than one block on another. We might want to include a deductive rule which determines that if no block is on top of a particular block then that block is clear. This rule would usually be applied after a block has been moved from one location to another. This is not equivalent to simply adding Clear(x) to the postcondition of the move-block(y,x,z) operator, since not all block moves result in x becoming clear.

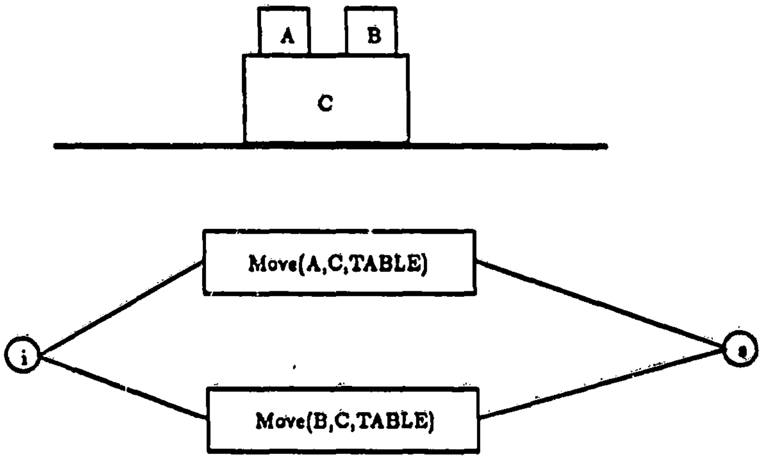


Figure 9.2: Deductive Plan

Take example 9.2. In this example each move operator has $On(x,y)$ in the delete list, where x is the block being moved. However, it does not have $Clear(y)$ in the add list, since it might be the case that there are remaining blocks on y. To determine if a block y is clear the planner must use the following axiom $(\neg\exists x\ On(x,y)) \Rightarrow Clear(y)$.

In our example, it is the case that $clear(C)$ holds in state S. The Modal Truth Criterion would not recognize this since no action prior to s necessarily asserts $clear(C)$.

In what follows we present a planning formalism which generalizes most of the powerful plan representations proposed in the literature to date. We then present a

Classical Truth Criterion which is proven valid for this formalism, and as a consequence, the formalisms which it subsumes.

9.3 The Classical Planning Ontology

In this section we present our ontology, upon which we will base our logic, and present the truth criterion.

A *plan* consists of the following components:

A $\{a_1, ..., a_n\}$, a set of n actions.

P A possibly infinite set of predicates.

W_< A partial order $A \times A$.

W_asserts A binary relation $A \times P$.

W_denies A binary relation $A \times P$.

W_initially A unary relation on P.

W_ground A unary relation on P.

W_holds A binary relation $P \times A$.

Notice that we have said nothing about the structure of actions or predicates for that matter, only that they exist. It is important to make the distinction at this point between the predicates in P, which are predicates on the particular domain the planner is operating in, e.g. ON(x,y), CLEAR(x) in the blocks world, and the predicates in the logical language used to describe the truth criterion, e.g. asserts(W,r) in Chapman's logic.

A *model* is a Kripke structure such that:

- The worlds are plans.
- A, P, W_initially, and W_ground do not vary from world to world.

We define some worlds as being *GLP* (ground-linear plans):

Definition 7 (*GLP*) $GLP(w)$ iff in world w, W_< defines a total order on A and for every predicate p such that there is some action a and some pair $(a,p) \in$ (W_asserts $\cup$ W_denies) it is the case that $p \in$ W_ground.

Further, the worlds in our Kripke structure are related by the following two relations:

Definition 8 (S) $S(w_1, w_2)$ is any reflexive, transitive relation such that whenever $GLP(w_1)$ then $S(w_1, w_2) \equiv (w_1 = w_2)$.

Definition 9 (C) $C(w_1, w_2)$ iff $S(w_1, w_2)$ and $GLP(w_2)$.

This completes our ontology. Notice that the relation S is not completely specified. The point is that any relation with the necessary properties we have defined will be adequate for our purposes. The relation S will vary from planner to planner, depending on the planner's particular method of specializing plans.

Further notice that this ontology can be used to represent planning formalisms as diverse as STRIPS, TWEAK, SIPE, NOAH, NONLIN, etc. This is because we impose no constraints on the structure of actions, and allow the relations W_<, W_asserts, W_denies, to vary from world to world, thus allowing for nonlinearity, conditional actions, deductive rules, etc.

9.4 Classical Plan Logic

In order to reason with our ontology we need a language which will allow us to make precise statements about our model. Not surprisingly, we will use a first-order modal logic. We define symbols for the members of A and P. For simplicity of exposition we will use the same symbol names in our logic as in the model; if $a \in A$ in the model then a is an action in our logic. We then define the following relations:

$\prec$ $(a_1 \prec a_2)$ iff $(a_1, a_2) \in$ W_<.

asserts asserts(a,p) iff $(a,p) \in$ W_asserts.

denies denies(a,p) iff $(a,p) \in$ W_denies.

initially initially(p) iff $p \in$ W_initially.

ground ground(p) iff $p \in$ W_ground.

holds holds(p,a) iff $(p,a) \in$ W_holds.

We intend holds(p,a) to be true if predicate p is true just before action a is executed; initially(p) is true if p is true in the initial situation; asserts(a,p) is true if action a asserts predicate p; denies(a,p) is true if action a denies predicate p; ground(p) is true if predicate p contains no variables in its argument list; $a_1 \prec a_2$ is true if action a_1 must be executed before action a_2.

We also have two sets of modal operators: $\Box_S, \Diamond_S$ defined in the usual manner on the accessibility relation S, and $\Box, \Diamond$ defined on the accessibility relation C.

We will need the following properties of our logic:

$\Box(P \wedge Q) \equiv \Box P \wedge \Box Q$

$\Diamond(P \vee Q) \equiv \Diamond P \vee \Diamond Q$

$\Box P \wedge \Diamond Q \Rightarrow \Diamond(P \wedge Q)$

$\Box_S(P \wedge Q) \equiv \Box_S P \wedge \Box_S Q$

$\Diamond_S(P \vee Q) \equiv \Diamond_S P \vee \Diamond_S Q$

These are true in all standard models of modal logic and therefore also in ours.

$\forall x \Box P \Rightarrow \Box \forall x P$ (Barcan Formula)

$\exists x \Box P \Rightarrow \Box \exists x P$

$\forall x \Box_S P \Rightarrow \Box_S \forall x P$ (Barcan Formula)

$\exists x \Box_S P \Rightarrow \Box_S \exists x P$

This follows from the fact that we are using a fixed domain, i.e. the objects in our domain (actions and predicates) do not vary from world to world. We can prove the Barcan formula as follows. If, in a plan α, $\forall x \Box P$ then it is the case that $\Box P$ is true in α no matter what value x takes. But then for all plans β such that $\alpha C \beta$ it is the case that P is true no matter what value x takes. But this means that $\forall x P$

is true in plan β. Which in turn means that $\Box\forall x P$ is true in plan α. [2] The proof for $\exists x \Box P \Rightarrow \Box\exists x P$ is similarly straightforward, and the proofs for $\Box_S$ and $\Diamond_S$ are analogous.

$\Diamond_S \Diamond P \Rightarrow \Diamond P$

If $\models_\alpha \Diamond_S \Diamond P$ then there exists some plan α' and some plan β such that $\alpha S \alpha'$ and $\alpha' C \beta$ and $\models_\beta P$.[3] We just need to show that $\alpha C \beta$. But this is clearly true, since $\alpha' C \beta$ implies $\alpha' S \beta$, which by transitivity implies $\alpha S \beta$. Furthermore, $GLP(\beta)$ which implies $\alpha C \beta$.

$\Box_S P \Rightarrow \Box P$

This follows directly from the definitions of C and S. Specifically, for all plans α and β whever $\alpha C \beta$ it is the case that $\alpha S \beta$. Therefore, if $\models_\omega P$ for every ω such that $\alpha S \omega$, it must be the case that $\models_\zeta P$ for every ζ such that $\alpha C \zeta$.

$\Box(\Diamond P \equiv P)$

$\Box(\Box P \equiv P)$

These two axioms follow from the fact that the only ground linear completion of a ground linear completion α is α. Therefore $\models_\alpha \Diamond P$ is equivalent to $\models_\alpha P$ and $\models_\alpha \Box P$ is equivalent to $\models_\alpha P$.

9.4.1 STRIPS Assumption

We are almost done with the definition of our logic. It turns out that to prove our Classical Truth Criterion we need one axiom:

$$\begin{gathered}\Box(\text{Holds}(p,a) \equiv \\ (\neg\exists b(b \prec a) \wedge \text{initially}(p)) \vee \\ \exists c((c \prec a) \wedge \neg\exists d((c \prec d) \wedge (d \prec a)) \wedge \\ (\text{asserts}(c,p) \vee (\text{Holds}(p,c) \wedge \neg\text{denies}(c,p)))))\end{gathered} \tag{9.1}$$

Interestingly enough, if we take the STRIPS assumption to be:

[2]For a discussion of the Barcan formula see [Hughes and Cresswell, 1968] pp. 147-148.
[3]The notation $\models_\alpha P$ means that $\models P$ in plan (world) α.

The truth value of a predicate does not change unless it is explicitly asserted or denied by an action in the plan.

Then it should be clear that our axiom is merely a restatement of this principle in our logic. Therefore, we will refer to this axiom as the *STRIPS Assumption Axiom*.

9.4.2 Lemmas

In our proof of the Classical truth criterion we will need the following lemmas which follow directly from the STRIPS Assumption Axiom.

Lemma 9.4.1

$$\Box((\neg\exists b\ (b \prec a) \wedge \neg\text{Initially}(p)) \Rightarrow \neg\text{Holds}(p,a))$$

Lemma 9.4.2

$$\Box((\exists b\ (b \prec a) \wedge \neg\exists c\ ((b \prec c) \wedge (c \prec a)) \wedge \neg\text{Holds}(p,b) \\ \wedge\neg\text{asserts}(b,p)) \Rightarrow \neg\text{Holds}(p,a))$$

Lemma 9.4.3

$$\Box((\exists b(b \prec a) \wedge \neg\exists c((b \prec c) \wedge (c \prec a)) \wedge \text{denies}(b,p) \wedge \neg\text{asserts}(b,p)) \\ \Rightarrow \neg\text{Holds}(p,a))$$

Lemma 9.4.4

$$\Box(\text{Initially}(p) \Rightarrow (\text{Holds}(p,a) \vee \exists b(b \prec a)))$$

Lemma 9.4.5

$$\Box(\exists b(b \prec a) \wedge \neg\exists c((b \prec c) \wedge (c \prec a))\wedge \\ (\neg\text{asserts}(b,p) \Rightarrow \text{Holds}(p,b) \wedge \neg\text{denies}(b,p)) \\ \Rightarrow \text{Holds}(p,a))$$

9.4.3 First attempt at defining a new truth criterion

One obvious truth criterion is the following formula:

$$\begin{aligned}
&\Box\text{Holds}(p,a) \equiv \\
&\Box((\neg\exists b(b \prec a) \wedge \text{initially}(p)) \vee \\
&\exists c((c \prec a) \wedge \neg\exists d((c \prec d) \wedge (d \prec a)) \wedge \\
&(\text{asserts}(c,p) \vee (\text{Holds}(p,c) \wedge \neg\text{denies}(c,p)))))
\end{aligned} \tag{9.2}$$

This follows directly from the STRIPS Assumption Axiom. It should be obvious that this definition is not particularly useful since it requires us to examine every ground linear completion of a plan to determine the truth of a proposition in a situation.

We now present a new truth criterion which is powerful enough to correctly handle an extended plan representation, yet allows us to determine the truth of a proposition more efficiently than simply examining every ground linear completion of the current plan.

9.5 Classical Truth Criterion

In our language the Classical Truth Criterion is expressed as follows:

$$\begin{aligned}
&\Box\ \text{Holds}(p,a) \equiv \\
&\quad \Box_S\ (((\neg\exists b\ \Diamond(b \prec a)) \Rightarrow \Box\text{Initially}(p)) \wedge \\
&\qquad \forall c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a))) \Rightarrow \\
&\qquad\quad (\neg\Diamond\ \text{asserts}(c,p) \Rightarrow \\
&\qquad\qquad \Box\text{Holds}(p,c) \wedge \neg\Diamond\text{denies}(c,p)))
\end{aligned}$$

Proof:

We will refer to the right hand side of the equivalence as TC. First we prove that if p holds before an action a of all ground-linear completions of a plan then the truth criterion must hold for that plan. This is done by proving the contrapositive, namely that $\neg TC(p,a) \Rightarrow \neg\Box\text{Holds}(p,a)$.

$$\begin{gathered}\Diamond_S((\neg\exists b\ \Diamond(b \prec a) \wedge \neg\Box\mathrm{Initially}(p)) \\ \vee\exists c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ (\neg\Box\mathrm{Holds}(p,c) \vee \Diamond\mathrm{denies}(c,p)))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a)\end{gathered} \tag{9.3}$$

Using $\Diamond_S(P \vee Q) \equiv \Diamond_S P \vee \Diamond_S Q$ we can rewrite the above.

$$\begin{gathered}\Diamond_S((\neg\exists b\ \Diamond(b \prec a) \wedge \neg\Box\mathrm{Initially}(p))) \\ \vee\Diamond_S(\exists c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ (\neg\Box\mathrm{Holds}(p,c) \vee \Diamond\mathrm{denies}(c,p)))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a)\end{gathered} \tag{9.4}$$

Using $P \vee Q \Rightarrow R \equiv (P \Rightarrow R) \wedge (Q \Rightarrow R)$:

$$\begin{gathered}\Diamond_S((\neg\exists b\ \Diamond(b \prec a) \wedge \neg\Box\mathrm{Initially}(p))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a) \\ \wedge \\ \Diamond_S(\exists c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ (\neg\Box\mathrm{Holds}(p,c) \vee \Diamond\mathrm{denies}(c,p)))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a)\end{gathered} \tag{9.5}$$

We first show:

$$\begin{gathered}\Diamond_S((\neg\exists b\ \Diamond(b \prec a) \wedge \neg\Box\mathrm{Initially}(p))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a)\end{gathered} \tag{9.6}$$

Since $\neg\exists b\Diamond(b \prec a) \equiv \Box\neg\exists b(b \prec a)$ we can rewrite the antecedent:

$$\Diamond_S(\Box\neg\exists b\ (b \prec a) \wedge \neg\Box\mathrm{Initially}(p)) \tag{9.7}$$

Now, since $\neg\Box P \equiv \Diamond\neg P$:

$$\Diamond_S(\Box\neg\exists b\ (b \prec a) \land \Diamond\neg \text{Initially}(p)) \tag{9.8}$$

Because $\Box P \land \Diamond Q \Rightarrow \Diamond(P \land Q)$:

$$\Diamond_S\Diamond(\neg\exists b\ (b \prec a) \land \neg \text{Initially}(p)) \tag{9.9}$$

Since $\Diamond_S\Diamond P \Rightarrow \Diamond P$:

$$\Diamond(\neg\exists b\ (b \prec a) \land \neg \text{Initially}(p)) \tag{9.10}$$

Using Lemma [illegible]4.1:

$$\Diamond\neg \text{Holds}(p, a) \tag{9.11}$$

Which is equivalent to $\neg\Box\text{Holds}(p, a)$ which is what we wanted to show. We still need to show part (2) of equation 9.5.

$$\begin{gathered}\Diamond_S(\exists c\ (\Box(c \prec a) \land \neg\exists d\ \Diamond((c \prec d) \land (d \prec a)) \\ \land\neg\Diamond\text{asserts}(c, p)\land \\ (\neg\Box\text{Holds}(p, c) \lor \Diamond\text{denies}(c, p)))) \\ \Rightarrow \neg\Box\text{Holds}(p, a)\end{gathered} \tag{9.12}$$

We distribute over $\lor$.

$$\begin{gathered}\Diamond_S(\exists c\ (\Box(c \prec a) \land \neg\exists d\ \Diamond((c \prec d) \land (d \prec a)) \\ \land\neg\Diamond\text{asserts}(c, p)\land \\ \neg\Box\text{Holds}(p, c))\lor \\ (\Box(c \prec a) \land \neg\exists d\ \Diamond((c \prec d) \land (d \prec a)) \\ \land\neg\Diamond\text{asserts}(c, p)\land \\ \Diamond\text{denies}(c, p))) \\ \Rightarrow \neg\Box\text{Holds}(p, a)\end{gathered} \tag{9.13}$$

We distribute $\exists c$ over $\vee$.

$$\begin{array}{c}\Diamond_S(\exists c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ \neg\Box\mathrm{Holds}(p,c))\vee \\ \exists c(\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ \Diamond\mathrm{denies}(c,p))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a)\end{array} \tag{9.14}$$

Using $\Diamond_S(P \vee Q) \equiv \Diamond_S P \vee \Diamond_S Q$

$$\begin{array}{c}\Diamond_S(\exists c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ \neg\Box\mathrm{Holds}(p,c)))\vee \\ \Diamond_S(\exists c(\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ \Diamond\mathrm{denies}(c,p))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a)\end{array} \tag{9.15}$$

Using $P \vee Q \Rightarrow R \equiv (P \Rightarrow R) \wedge (Q \Rightarrow R)$:

$$\begin{array}{c}\Diamond_S(\exists c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ \neg\Box\mathrm{Holds}(p,c))) \Rightarrow \neg\Box\mathrm{Holds}(p,a) \\ \vee \\ \Diamond_S(\exists c(\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathrm{asserts}(c,p)\wedge \\ \Diamond\mathrm{denies}(c,p))) \\ \Rightarrow \neg\Box\mathrm{Holds}(p,a)\end{array} \tag{9.16}$$

We now show:

$$\begin{gathered}\Diamond_S(\exists c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\mathsf{asserts}(c,p)\wedge \\ \neg\Box\mathsf{Holds}(p,c))) \Rightarrow \neg\Box\mathsf{Holds}(p,a)\end{gathered} \tag{9.17}$$

$\exists\ x\Diamond P \Rightarrow \Diamond\exists\ xP$ and $\neg\Diamond P \equiv \Box\neg P$.

$$\begin{gathered}\Diamond_S(\exists c\ (\Box(c \prec a) \wedge \Box\neg\exists d\ ((c \prec d) \wedge (d \prec a)) \\ \wedge\Box\neg\mathsf{asserts}(c,p)\wedge \\ \Diamond\neg\mathsf{Holds}(p,c)))\end{gathered} \tag{9.18}$$

$\Box P \wedge \Diamond Q \Rightarrow \Diamond(P \wedge Q)$

$$\begin{gathered}\Diamond_S(\exists c\ \Diamond((c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\mathsf{asserts}(c,p)\wedge \\ \neg\mathsf{Holds}(p,c)))\end{gathered} \tag{9.19}$$

$\exists c\ \Diamond P(c) \Rightarrow \Diamond\exists c\ P(c)$:

$$\begin{gathered}\Diamond_S\Diamond(\exists c\ (c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (\underline{d \prec} a)) \\ \wedge\neg\mathsf{asserts}(c,p)\wedge \\ \neg\mathsf{Holds}(p,c))\end{gathered} \tag{9.20}$$

$\Diamond_S\Diamond P \Rightarrow \Diamond P$:

$$\begin{gathered}\Diamond(\exists c\ (c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\mathsf{asserts}(c,p)\wedge \\ \neg\mathsf{Holds}(p,c))\end{gathered} \tag{9.21}$$

Applying Lemma 9.4.2:

$$\Diamond\neg\mathsf{Holds}(p,a) \tag{9.22}$$

Which is equivalent to $\neg\Box\mathsf{Holds}(p,a)$.

We still need to show:

$$\begin{gathered}\Diamond_S(\exists c(\Box(c \prec a) \wedge \neg\exists d\,\Diamond((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\Diamond\text{asserts}(c,p)\wedge \\ \Diamond\text{denies}(c,p))) \\ \Rightarrow \neg\Box\text{Holds}(p,a)\end{gathered} \tag{9.23}$$

$\neg\Diamond P \Rightarrow \Box\neg P$.

$$\begin{gathered}\Diamond_S(\exists c(\Box(c \prec a) \wedge \Box\neg\exists d\ ((c \prec d) \wedge (d \prec a)) \\ \wedge\Box\neg\text{asserts}(c,p)\wedge \\ \Diamond\text{denies}(c,p)))\end{gathered} \tag{9.24}$$

$\Box P \wedge \Diamond Q \Rightarrow \Diamond(P \wedge Q)$:

$$\begin{gathered}\Diamond_S(\exists c\Diamond((c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\text{asserts}(c,p)\wedge \\ \text{denies}(c,p)))\end{gathered} \tag{9.25}$$

$\exists a\Diamond P(a) \Rightarrow \Diamond\exists a P(a)$

$$\begin{gathered}\Diamond_S\Diamond(\exists c(c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\text{asserts}(c,p)\wedge \\ \text{denies}(c,p))\end{gathered} \tag{9.26}$$

$\Diamond_S\Diamond P \Rightarrow \Diamond P$

$$\begin{gathered}\Diamond(\exists c(c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a)) \\ \wedge\neg\text{asserts}(c,p)\wedge \\ \text{denies}(c,p))\end{gathered} \tag{9.27}$$

Using Lemma 9.4.3:

$$\Diamond\neg\text{Holds}(p,a) \tag{9.28}$$

Which is equivalent to $\neg\Box\text{Holds}(p,a)$. We are now done proving $\Box\text{Holds}(p,a) \Rightarrow TC$. We now show $TC \Rightarrow \Box\text{Holds}(p,a)$.

$$\begin{aligned}\Box_S\ &(((\neg\exists b\ \Diamond(b \prec a)) \Rightarrow \Box\text{Initially}(p))\wedge \\ &\forall c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a))) \Rightarrow \\ &\quad(\neg\Diamond\ \text{asserts}(c,p) \Rightarrow \\ &\qquad\Box\text{Holds}(p,c) \wedge \neg\Diamond\text{denies}(c,p)))\end{aligned}$$

$\Box_S(P \wedge Q) \Rightarrow \Box_S P \wedge \Box_S Q$:

$$\begin{aligned}\Box_S\ &((\neg\exists b.\Diamond(b \prec a)) \Rightarrow \Box\text{Initially}(p))\wedge \\ &\Box_S(\forall c\ (\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a))) \Rightarrow \\ &\quad(\neg\Diamond\ \text{asserts}(c,p) \Rightarrow \\ &\qquad\Box\text{Holds}(p,c) \wedge \neg\Diamond\text{denies}(c,p)))\end{aligned}$$

We begin by showing:

$$\begin{aligned}&\Box_S((\neg\exists b\ \Diamond(b \prec a)) \Rightarrow \Box\text{Initially}(p)) \\ &\quad\Rightarrow \Box(\text{Holds}(p,a) \vee \exists b(b \prec a))\end{aligned} \tag{9.29}$$

$\exists b\Diamond(b \prec a) \Rightarrow \Diamond\exists b(b \prec a)$:

$$\Box_S((\neg\Diamond\exists b\ (b \prec a)) \Rightarrow \Box\text{Initially}(p)) \tag{9.30}$$

$\neg\Diamond P \Rightarrow \Box\neg P$.

$$\Box_S((\Box\neg\exists b\ (b \prec a)) \Rightarrow \Box\text{Initially}(p)) \tag{9.31}$$

$\Box_S P \Rightarrow \Box P$:

$$\Box((\Box\neg\exists b\ (b \prec a)) \Rightarrow \Box\text{Initially}(p)) \tag{9.32}$$

$\Box(\Box P \equiv P)$:

$$\Box((\neg\exists b\ (b \prec a)) \Rightarrow \text{Initially}(p)) \tag{9.33}$$

Rewriting,

$$\Box((\exists b\ (b \prec a)) \vee \text{Initially}(p)) \tag{9.34}$$

Using lemma 9.4.4:

$$\Box(\text{Initially}(p) \Rightarrow (\exists b(b \prec a)) \vee \text{Holds}(p,a)) \tag{9.35}$$

Therefore:

$$\Box((\exists b\ (b \prec a)) \vee \text{Holds}(p,a)) \tag{9.36}$$

We now show:

$$\begin{gathered}\Box_S(\forall c(\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a))) \Rightarrow \\ (\neg\Diamond\text{asserts}(c,p) \Rightarrow (\Box\text{Holds}(p,c) \wedge \neg\Diamond\text{denies}(c,p)))) \\ \Rightarrow \Box(\text{Holds}(p,a) \vee \neg\exists e(e \prec a))\end{gathered} \tag{9.37}$$

$\neg\Diamond P \equiv \Box\neg P$ and $\exists x\Diamond P \Rightarrow \Diamond\exists x P$.

$$\begin{gathered}\Box_S(\forall c(\Box(c \prec a) \wedge \Box\neg\exists d\ ((c \prec d) \wedge (d \prec a))) \Rightarrow \\ (\Box\neg\text{asserts}(c,p) \Rightarrow (\Box\text{Holds}(p,c) \wedge \Box\neg\text{denies}(c,p))))\end{gathered} \tag{9.38}$$

$\Box_S P \Rightarrow \Box P$:

$$\begin{gathered}\Box(\forall c(\Box(c \prec a) \wedge \Box\neg\exists d\ ((c \prec d) \wedge (d \prec a))) \Rightarrow \\ (\Box\neg\text{asserts}(c,p) \Rightarrow (\Box\text{Holds}(p,c) \wedge \Box\neg\text{denies}(c,p))))\end{gathered} \tag{9.39}$$

$\Box(\Box P \equiv P)$:

$$\begin{gathered}\Box(\forall c((c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a))) \Rightarrow \\ (\neg\text{asserts}(c,p) \Rightarrow (\text{Holds}(p,c) \wedge \neg\text{denies}(c,p))))\end{gathered} \tag{9.40}$$

$(\exists e(e \prec a) \vee \neg\exists e(e \prec a)) \wedge P \Rightarrow (\exists e(e \prec a) \wedge P) \vee \neg\exists e(e \prec a)$.

$$\begin{gathered}\Box((\exists e(e \prec a) \wedge \forall c((c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a))) \Rightarrow \\ (\underline{\neg\text{asserts}}(c,p) \Rightarrow (\text{Holds}(p,c) \wedge \neg\text{denies}(c,p)))) \\ \vee(\neg\exists e(e \prec a)))\end{gathered} \tag{9.41}$$

Since a ground linear plan is a total order the following holds:

$$\Box(\exists e(e \prec a) \equiv \exists f(f \prec a) \wedge \neg\exists g(f \prec g) \wedge (g \prec a))$$

Therefore:

$$\Box((\exists f(f \prec a) \wedge \neg\exists g((f \prec g) \wedge (g \prec a)) \wedge \forall c((c \prec a) \wedge \neg\exists d\ ((c \prec d) \wedge (d \prec a))) \Rightarrow \\ (\neg\text{asserts}(c,p) \Rightarrow (\text{Holds}(p,c) \wedge \neg\text{denies}(c,p)))) \\ \vee(\neg\exists e(e \prec a))) \tag{9.42}$$

We use universal instantiation and simplify:

$$\Box((\exists f(f \prec a) \wedge \neg\exists g((f \prec g) \wedge (g \prec a)) \wedge \\ (\neg\text{asserts}(f,p) \Rightarrow (\text{Holds}(p,f) \wedge \neg\text{denies}(f,p)))) \\ \vee(\neg\exists e(e \prec a))) \tag{9.43}$$

Using lemma 9.4.5:

$$\Box(\text{Holds}(p,a) \vee (\neg\exists e(e \prec a))) \tag{9.44}$$

Therefore:

$$\Box(\text{Holds}(p,a) \vee \exists b(b \prec a)) \wedge \Box(\text{Holds}(p,a) \vee \neg\exists e(e \prec a)) \tag{9.45}$$

$\Box P \wedge \Box Q \Rightarrow \Box(P \wedge Q)$:

$$\Box((\text{Holds}(p,a) \vee \exists b(b \prec a)) \wedge (\text{Holds}(p,a) \vee \neg\exists e(e \prec a))) \tag{9.46}$$

Rewriting:

$$\Box(\text{Holds}(p,a) \vee (\exists b(b \prec a) \wedge \neg\exists e(e \prec a))) \tag{9.47}$$

Which is equivalent to:

$$\Box\text{Holds}(p,a) \tag{9.48}$$

$\Box$ Q.E.D.

This completes our proof of the Classical Truth Criterion.

9.6 Algorithm for checking truth criterion

We can make use of two properties of the truth criterion to increase the efficiency of a truth checking algorithm. The first is that if the truth criterion is non-trivially true in a plan, then it is true for all specializations of it.

By non-trivially true we mean that either a is the first action of the plan and $\Box$Initially(p), or there is some action c immediately before a and

$$\neg\Diamond\text{asserts}(c,p) \wedge \Box\text{Holds}(p,c) \wedge \neg\Diamond\text{denies}(c,p).$$

It is important to note that if there is some action c immediately before a there can be no other action immediately before a. This property guarantees that both conditions imply the Classical Truth Criterion.

We show,

$$\begin{gathered}(((\neg\exists b\ \Diamond(b \prec a)) \wedge \Box\text{Initially}(p))\vee\\ \exists c(\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)))\wedge\\ (\neg\Diamond\text{asserts}(c,p) \wedge \Box\text{Holds}(p,c) \wedge \neg\Diamond\text{denies}(c,p)))\\ \Rightarrow\\ \Box_S(((\neg\exists b\ \Diamond(b \prec a)) \wedge \Box\text{Initially}(p))\vee\\ \exists c(\Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)))\wedge\\ (\neg\Diamond\text{asserts}(c,p) \wedge \Box\text{Holds}(p,c) \wedge \neg\Diamond\text{denies}(c,p)))\end{gathered} \tag{9.49}$$

Proof:

Rewrite the antecedent:

$$\begin{gathered}(((\forall b\ \Box\neg(b \prec a)) \wedge \Box\text{Initially}(p))\vee\\ \exists c(\Box(c \prec a) \wedge \forall d\ \Box\neg((c \prec d) \wedge (d \prec a)))\wedge\\ (\Box\neg\text{asserts}(c,p) \wedge \Box\text{Holds}(p,c) \wedge \Box\neg\text{denies}(c,p)))\end{gathered} \tag{9.50}$$

Use $\Box P \Rightarrow \Box_S\Box P$:

$$\begin{gathered}(((\forall b\ \Box_S\Box\neg(b \prec a)) \wedge \Box_S\Box\text{Initially}(p))\vee\\ \exists c(\Box_S\Box(c \prec a) \wedge \forall d\ \Box_S\Box\neg((c \prec d) \wedge (d \prec a)))\wedge\\ (\Box_S\Box\neg\text{asserts}(c,p) \wedge \Box_S\Box\text{Holds}(p,c) \wedge \Box_S\Box\neg\text{denies}(c,p)))\end{gathered} \tag{9.51}$$

Use the Barcan formula:

$$\begin{gathered}(((\Box_S \forall b\ \Box\neg(b \prec a)) \wedge \Box_S \Box \mathrm{Initially}(p)) \vee \\ \exists c(\Box_S \Box(c \prec a) \wedge \Box_S \forall d \Box\neg((c \prec d) \wedge (d \prec a))) \wedge \\ (\Box_S \Box\neg \mathrm{asserts}(c,p) \wedge \Box_S \Box \mathrm{Holds}(p,c) \wedge \Box_S \Box\neg \mathrm{denies}(c,p)))\end{gathered} \tag{9.52}$$

Use $\Box_S P \wedge \Box_S Q \Rightarrow \Box_S(P \wedge Q)$:

$$\begin{gathered}\Box_S((\forall b\ \Box\neg(b \prec a)) \wedge \Box \mathrm{Initially}(p)) \vee \\ \exists c \Box_S(\Box(c \prec a) \wedge \forall d \Box\neg((c \prec d) \wedge (d \prec a))) \wedge \\ (\Box\neg \mathrm{asserts}(c,p) \wedge \Box \mathrm{Holds}(p,c) \wedge \Box\neg \mathrm{denies}(c,p)))\end{gathered} \tag{9.53}$$

Use the Barcan formula:

$$\begin{gathered}\Box_S((\forall b\ \Box\neg(b \prec a)) \wedge \Box \mathrm{Initially}(p)) \vee \\ \Box_S \exists c(\Box(c \prec a) \wedge \forall d \Box\neg((c \prec d) \wedge (d \prec a))) \wedge \\ (\Box\neg \mathrm{asserts}(c,p) \wedge \Box \mathrm{Holds}(p,c) \wedge \Box\neg \mathrm{denies}(c,p)))\end{gathered} \tag{9.54}$$

$\Box_S P \vee \Box_S Q \Rightarrow \Box_S(P \vee Q)$

$$\begin{gathered}\Box_S(((\forall b\ \Box\neg(b \prec a)) \wedge \Box \mathrm{Initially}(p)) \vee \\ \exists c(\Box(c \prec a) \wedge \forall d \Box\neg((c \prec d) \wedge (d \prec a))) \wedge \\ (\Box\neg \mathrm{asserts}(c,p) \wedge \Box \mathrm{Holds}(p,c) \wedge \Box\neg \mathrm{denies}(c,p)))\end{gathered} \tag{9.55}$$

Rewrite:

$$\begin{gathered}\Box_S(((\neg\exists b \Diamond(b \prec a)) \wedge \Box \mathrm{Initially}(p)) \vee \\ \exists c(\Box(c \prec a) \wedge \neg\exists d \Diamond((c \prec d) \wedge (d \prec a))) \wedge \\ (\neg\Diamond \mathrm{asserts}(c,p) \wedge \Box \mathrm{Holds}(p,c) \wedge \neg\Diamond \mathrm{denies}(c,p)))\end{gathered} \tag{9.56}$$

Q.E.D.

The other is that only plans in which the following holds can possibly fail the truth criterion:

$$\begin{gathered}\neg\exists b\ \Diamond(b \prec a) \vee \\ \exists c \Box(c \prec a) \wedge \neg\exists d\ \Diamond((c \prec d) \wedge (d \prec a)) \wedge \neg\Diamond \mathrm{asserts}(c,p)\end{gathered}$$

An algorithm for checking the truth criterion then only has to examine the "weakest" specializations such that the above condition holds to see if $\Box \mathrm{holds}(p,c) \wedge$

```
Holds(p,a)
    For every distinct, minimal specialization such that there is an action c
    immediately before a and c does not possibly assert p
            check that c does not possibly deny p and necessarily Holds(p,c).
    If there is a minimal specialization such that a is the first action
            check that initially(p) necessarily holds.
```

Table 9.1: Algorithm for checking truth criterion.

$\neg\Diamond$denies(c,p). The details of the algorithm will vary from planner to planner depending on their underlying plan representation, however a high-level description of it can be found in table 9.1. A specialization w_1 satisfying a condition is *minimal* if there is no other specialization w_2 satisfying that condition such that $S(w_2, w_1)$.

This algorithm does not necessarily require that every ground linear completion of the plan be checked. Suppose we are given the plan in figure 9.3. In this example we assume the TWEAK planning formalism, extended with restricted ranges on variables. We are interested in knowing whether the precondition, P(A,B) of action 3 necessarily holds, given that $x \in \{A, B\}$, $y \in \{A, B\}$ and $x \not\approx y$. In this case the algorithm will only have to check two specializations of the plan, namely the ones where $y \not\approx A$ and $x \not\approx B$, which are added when we guarantee that action 2 not assert P(A,B). The reason we do not have to check more specializations is that we cannot generate any specializations such that action 1 does not assert P(A,B), given that action 2 cannot assert P(A,B) either. This is much more efficient than having to check every possible ground linear completion of this plan.

However, we are still left with the problems of the efficiency of constraint propagation. We might ask how planners with extended representational capabilities deal with these problems. SIPE is the best example of a planner with powerful representational capabilities. Furthermore, SIPE is the most efficient planner to date.

In SIPE, the combinatorial nature of constraint propagation is handled by keeping many of the constraint propagations local, i.e. not performing a global constraint check every time a constraint is posted. Furthermore, constraint computations are not performed every time a constraint is added, but rather at regular intervals. This

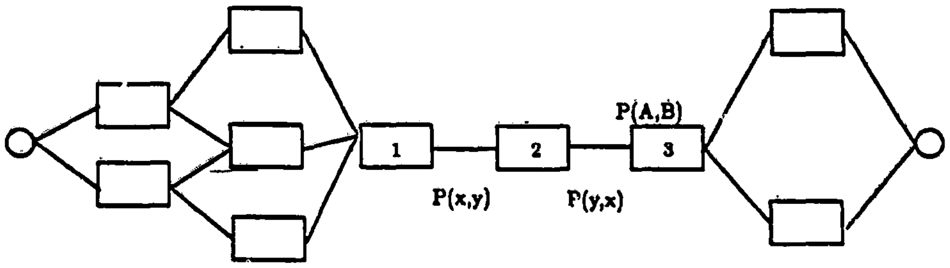


Figure 9.3: A plan with restricted ranges on variables

seems to be an adequate compromise, as the constraint computations have seldom led to a problem [Wilkins, 1988].

State independent axioms are handled by computing their derived effects only when a new action is inserted into the plan. The propositions that are derived in this manner are inserted in the add list of the action that is being inserted. By using this method, checking whether an action asserts a proposition is done trivially by checking if it codesignates with a proposition in the add list. Of course this can lead to inconsistencies later in planning, but this has not proven to be a serious problem with SIPE.

Relating this method to our truth criterion we can see that it would greatly speed up the computation everywhere we need to check if a particular situation asserts a proposition. Although it requires the addition of unsound methods, experience seems to bear out that a richer representation makes up for the potential drawbacks.

9.7 Summary

In this chapter we presented a new truth criterion for a powerful plan formalism which subsumes most planners proposed to date. In fact, the only axiom in our formalism is a restatement of the STRIPS assumption. We proved it is sound and complete with respect to this representation. We then showed how it gives rise to an algorithm that is more efficient than simply checking every possible completion of a plan.

Chapter 10

Further Work

In this chapter we point out some questions raised by this thesis and propose further work that might help resolve these.

10.1 Real World Applications

The most important question that needs to be answered is whether predicate relaxation can scale up to real world applications. Since the work presented in this thesis was based on a complete planner, real world applications were not within its scope.

One way to answer this is to apply predicate relaxation to an efficient planner. SIPE [Wilkins, 1984] is an obvious candidate. In the early stages of this research SIPE was used for some experiments with good results. We hope to some day attempt to integrate predicate relaxation into SIPE.

10.2 Extensions to Predicate Relaxation

Another interesting avenue to pursue is extending predicate relaxation. This might involve trading off correctness for simplicity in generating the relaxed predicates. As the efficiency of the planning process is improved it is likely that the complexity of the relaxed predicates becomes the bottleneck. This then leads us to new ways of simplifying relaxed predicates, so that they can be evaluated more quickly.

Another extension would be to define a new form of predicate relaxation which can take into account the more complex operators available in state of the art planners. In this thesis we have assumed a planner which is based on STRIPS-style operators. It would_be interesting to come up with a definition which can handle conditional operators, take into account resources, etc.

There is also the possibility of defininining *formula relaxation*, namely the relaxation of commonly occuring sub-formulas. This idea has been proposed by Nilsson and might result in definitions that are similar to *tree plans* [Nilsson, 1989].

10.3 Implementation of the Classical Truth Criterion

Another interesting project might be to build a planner based on the Classical Truth Criterion. This planner would be able to support an extended language for representing operators, including conditional operators, as well as restricted domains for variables.

It would be interesting to experiment with different ways of trading off completeness and correctness for efficiency in the truth criterion. The new truth criterion would provide a good basis for performing such experiments.

10.4 Real-time Extensions

Another area that should be examined is that of improving real-time performance. The current extension to predicate relaxation allows a limited form of reactivity, since the planner is able to propose a plausible action in case it is interrupted. One goal of planning research should be to imbue planners with the capability to decide when acting is indeed necessary, rather than having to rely on an outside agent to make that decision. This is also closely tied to the problem of determining the quality of the currently chosen action, since the tradeoff between the improvement of actions resulting from future planning and the possible gains of acting immediately is one the planner should be aware of.

10.5 Operator Abstraction

More work needs to be done in the field of operator abstraction. This might entail automating the process of constructing abstract operators a la Nonlin and Sipe, but might also involve designing new forms of operator abstraction, utilizing approximations of operators, and automatically abstracting preconditions and effects of operators. Operator abstraction has been the prevalent form of abstraction in planning and automating the process would be a significant advance.

10.6 Concluding Remarks

This thesis has explored new methods of performing abstraction in planning. We hope that the usefulness of abstraction has been clarified, and that the methods, primarily predicate relaxation, provide a basis for future research in this area. State abstraction has been somewhat overlooked in the planning literature, but is beginning to see some resurgence. Furthermore, we stress the importance of exploring ways in which to improve the problem solving capabilities of planners as opposed to their representational prowess, an area that has already been much explored. Abstraction and planning remain fascinating areas of research that need to be further investigated.

Bibliography

[Agre and Chapman, 1987] Agre, P. and Chapman, D., "Pengi: An Implementation of a Theory of Activity" *AAAI-87*, Morgan Kaufmann, Los Altos, California.

[Amarel, 1968] Amarel, S., "On Representations of Problems of Reasoning About Actions," *Machine Intelligence* 3, D. Michie (ed.), pp 131-171, Edinburgh University Press, 1968.

[Allen et al, 1990] Allen, J., Hendler, J., and Tate, A. (Eds.), "Readings in Planning," Morgan Kaufman, San Mateo, 1990.

[Brooks, 1986] Brooks, R. A., "A Robust Layered Control System for a Mobile Robot," IEEE Journal of Robotics and Automation, Vol RA-2, No. 1, March 1986.

[Chapman and Agre, 1985] Chapman, D. and Agre, P. E., "Abstract Reasoning as Emergent from Concrete Activity," *Workshop on Planning and Reasoning about Action*, Portland, Oregon, 1986.

[Chapman, 1987] Chapman, D., "Planning for Conjunctive Goals," *Artificial Intelligence*, v 32: 333-378, July 1987.

[Chapman, 1989] Chapman, D., "Penguins Can Make Cake", *AI Magazine*, vol. 10, no. 4, 1989.

[Dean and Boddy, 1988] Dean, T. and Boddy M., "An Analysis of Time-dependent Planning," in *Proceedings AAAI-88*, pages 49-54, 1988.

[Drummond and Currie, 1988] Drummond, M., and Currie, K., "Exploiting Temporal Coherence in Nonlinear Plan Construction," *Computation Intelligence*, in press.

[Drummond and Tate, 1989] Drummond, M. and Tate, A., "AI Planning: A Tutorial and Review," Technical Report *AIAI-TR-30*, University of Edinburgh, U.K., 1989.

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, 2(3-4): 189-208, 1971.

[Fikes et al, 1972] Fikes, R. E., Hart P., and Nilsson, N. J., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, 3(4): 251-288, 1972.

[Firby, 1987] Firby, R. J., "An Investigation into Reactive Planning in Complex Domains," in *Proceedings AAAI-87.*

[Georgeff, 1987] Georgeff, M. P., "Planning," *Annual Review of Computer Science*, v 2: 359-400, 1987.

[Ginsberg, 1989] Ginsberg, M.L., "Universal Planning: An (Almost) Universally Bad Idea," *AI Magazine*, vol. 10, no. 4, 1989.

[Green, 69] Green, C., "The Application of Theorem Proving to Question-Answering Systems," Ph.D. thesis, Stanford University, Stanford, 1969.

[Harary and Palmer, 1973] Harary, F. and Palmer, E. M., *Graphical Enumeration*, Academic Press, New York, New York, 1973.

[Hoare, 1969] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *CACM* 12, 10, 576-580, 583.

[Hughes and Cresswell, 1968] Hughes, G.E. and Cresswell, M.J., "An Introduction to Modal Logic," Methuen and Co, London, England.

[King, 1969] King, J.C., "A Program Verifier," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

[Knoblock, 1990] Knoblock, C., "A Theory of Abstraction for Hierarchical Planning," in D.P. Benjamin (Ed.), *Change of Representation and Inductive Bias*, Boston, MA: Kluwer, 1990.

[Korf, 1985] Korf, R.E., "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, 27:97-111, 1985.

[Korf, 1987] Korf, R.E., "Planning as Search: A Quantitative Approach," *Artificial Intelligence*, 33:65-88, 1987.

[Lifschitz, 1986] Lifschitz, V., "On the Semantics of STRIPS," *Proceedings of the Workshop on Reasoning about Actions and Plans*, Timberline, Oregon, 1986.

[Lowry, 1988] Lowry, M. R., "Invariant Logic: A Calculus for Problem Reformulation," *Proceedings of the Seventh National Conference on Artificial Intelligence*, Saint Paul, Minnesota, 1988.

[Manna, 1968] Manna, Z., "Termination of Algorithms," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1968.

[McCarthy and Hayes, 1969] McCarthy, J. and Hayes, P. J., "Some Philosophical Problems from the Viewpoint of Artificial Intelligence," in: Michie, D. (Ed.), *Machine Intelligence* 7, 1969.

[Nilsson, 1989] Nilsson, N. J., "Teleo-Reactive Agents", Unpublished Draft, Stanford Computer Science Department, 1989.

[Nilsson et al, 1990] Nilsson, N. J., Moore, R., Torrance, M. C., "ACTNET: An Action-Network Language and its Interpreter (A Preliminary Report)," Stanford Computer Science Department Draft Report, 1990.

[Rosenschein and Kaelbling, 1987] Rosenschein, S. J. and Kaelbling, L. P., "The Synthesis of Digital Machines with Provable Epistemic Properties," SRI Technical Note 412, Menlo Park, Ca, 1987.

[Sacerdoti, 1974] Sacerdoti, E., "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence*, v 5: 115-135, 1974.

[Sacerdoti, 1977] Sacerdoti, E., *A Structure for Plans and Behavior*, Elsevier, North-Holland, New York, 1977.

[Schoppers, 1987] Schoppers, M. J., "Universal Plans for Reactive Robots in Unpredictable Environments," in *Proceedings AAAI-87*.

[Shoham, 1989] Shoham, Y., "Time for Action: On the Relation between Time, Knowledge and Action," *Proceedings IJCAI-89*, Detroit, Michigan, 1989.

[Sussman, 1973] Sussman, G.J., "A Computational Model of Skill Acquisition," Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA.

[Tate, 1977] Tate, A., "Generating Project Networks," *Proceedings IJCAI-77*, Cambridge, Massachusetts, 1977, pp. 888-893.

[Tenenberg, 1988] Tenenberg, J. D., "Abstraction in Planning," Ph.D. Thesis, Technical Report 250, University of Rochester, Rochester, New York, 1988.

[Vere, 1983] Vere, S., "Planning in Time: Windows and Durations for Activities and Goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v 5: 246-267, 1983.

[Waldinger, 1977] Waldinger, R., "Achieving Several Goals Simultaneously," *Machine Intelligence* 8:94-136, Elcock E. and Michie D. (Eds.), Ellis Horwood, 1977.

[Washington, 1989] Washington, R., "Abstraction Planning in Real Time," Ph.D. Thesis Proposal, unpublished, 1989.

[Wilensky, 1980] Wilensky, R., "Meta-Planning," *Proceedings AAAI-80*, Stanford, California, 1980, pp. 334-336.

[Wilkins, 1984] Wilkins, D. E., "Domain-independent Planning: Representation and Plan Generation," *Artificial Intelligence*, v 22: 269-301, April 1984.

[Wilkins, 1988] Wilkins, D. E., *Practical Planning*, Morgan Kaufmann, San Mateo, California, 1988.

[Yang, 1989] Yang, Q., "Improving the Efficiency of Planning," Ph.D. Thesis, University of Maryland, 1989.